University of Zurich[UZH]

**Department of Informatics**

# Efficient Distributed Stream Processing: Optimization Approaches and Applications

Dissertation submitted to the Faculty of Economics,
Business Administration and Information Technology
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Lorenz Fischer
from Switzerland

approved in September 2015

at the request of
Prof. Abraham Bernstein, Ph.D.
Prof. Philippe Cudré-Mauroux, Ph.D.

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, September 16, 2015

Chairman of the Doctoral Board: Prof. Dr. Renato Pajarola

# Abstract

As more aspects of our daily lives are being computerized, ever larger amounts of data are being produced at ever greater speeds. In this data lies great value, and we need technologies that enable us to extract this value. This thesis is concerned with one type of technology that allows us to do this: Distributed Stream Processing Systems (DSPS) are systems consisting of many computers that jointly process, and hence extract value from, large amounts of data at high speeds.

This dissertation consists of three research projects that investigate two aspects of DSPS: In two projects, different approaches to increase the efficiency of DSPS were studied and in one project, the value of increased efficiency in stream processing was evaluated. All of these projects have been conducted on real computer systems and they are all of quantitative nature. In the first study, a graph partitioning algorithm was leveraged to schedule the workload within a DSPS. This reduced the communication load between hosts, while maintaining or increasing the throughput of the system. The second study was concerned with the auto-configuration of DSPS. We used a probabilistic black-box optimization strategy called Bayesian Optimization to increase throughput performance of DSPSs through configuration. In the third study, we investigated the value of increased efficiency of a DSPS. This was done by building a DSPS based entity ranking system and by evaluating the effect of timely data processing on the quality of the generated rankings.

# Zusammenfassung

Immer grössere Teile unseres Alltags funktionieren computergestützt, wodurch immer grössere Datenmengen mit immer höherer Geschwindigkeit generiert werden. Diese Daten beinhalten grosses Wertpotential und wir benötigen Technologien um diesen Wert aus ihnen zu extrahieren. Die vorliegene Dissertation beschäftigt sich mit einer Technologie, die eben dies zum Ziel hat: Verteilte Datenstromsysteme (VDSS) sind Computersysteme, in denen ein Verbund aus vielen Einzelsystemen gemeinsam grosse Mengen von Daten in Echtzeit verarbeitet und damit Wert aus diesen Daten extrahiert.

Die Dissertation besteht aus drei Forschungsprojekten, in welchen zwei Aspekte von VDSS untersucht werden: In zwei dieser Projekte werden unterschiedliche Vorgehensweisen untersucht um die Effizienz von VDSS zu erhöhen. In einem weiteren Projekt wird der Wert dieser gesteigerten Effizient untersucht. Alle diese Projekte wurden mit Hilfe echter Computersysteme durchgeführt und alle sind von quantitativer Natur. Im ersten Projekt wurde ein Graphpartitionierungsalgorithmus dazu verwendet, die Arbeit innerhalb eines VDSS zu verteilen. Dadurch konnte der Kommunikationsaufwand zwischen Maschinen im Verbund verringert und gleichzeitig die Datendurchsatzrate erhalten oder sogar erhöht werden. Für das zweite Projekt wurde der Aspekt der Systemkonfiguration untersucht. Dazu wurde Bayes'sche Optimierung – eine probabilistische Blackbox-Optimierungsstrategie – dazu verwendet, den Datendurchsatz von VDSS durch Konfiguratation zu erhöhen. Für das dritte Projekt wurde der Wert der gesteigerten Effizienz von VDSS untersucht. Dazu wurde ein mit VDSS-Technologie gebautes System zum rangieren von Objektbeziehungen gebaut und der Effekt von zeitnaher Datenverarbeitung auf die Qualität der erstellen Ranglisten evaluiert.

# Acknowledgements

I[1] would like to express my sincerest gratitude to my advisor and mentor, Avi, for his continued support and guidance. His optimism, creativity, and mental flexibility are exceptional and I am grateful for the many things that he has taught me.

I also would like to thank the many people who accompanied me on this journey. I learned a great deal from all of them: Esther Kaufmann deserves special thanks for encouraging me to take on this challenge. Thank you, Cosmin, for being a fun office mate, friend, and for the insightful discussions about research and other things. I enjoyed all of them very much. Thank you, Philip, for being a good friend and for your general positive attitude. Your thoughts and feedback were always very helpful. Thank you, Ela, for being a friend, Madam Mim of DDIS, and supplier of many missing commas. I feel very fortunate to have had the chance to work in a very international research group and I thank Michael, Bibek, Shen, Dani, Vera, Cosmin, and Ela for letting me glimpse their respective cultures. I would also like to thank all members of our research group and friends at the department for their support and for making my time here at IFI the pleasant experience that it was: Adrian, Amancio, Andi, Avi, Barbara, Bibek, Christian, Dani Spicar, Dani Strebel, Doro, Ela, Flory, Helen, HP, Iris, Jayalath, Jonas, Juk, Kathi, Khoa, Marc, Michael, Mike, Patrick de Boer, Patrick Minder, Sarah, Shen, Timo, Tobi, and Tom.

The support and encouragement from my family and friends has been indispensable. The greatest source of support and strength has been my friend and love, Rebekka. Without her, I would very likely not be writing these lines.

Finally, I want to thank the people of the Canton of Zurich, the University of Zurich, and the Department of Informatics for having given me the opportunity to work and study in such an outstanding environment.

[1] In this thesis, the first person singular is only used to specifically express personal opinions.

# Table of Contents

# Part I

# Synopsis

# Synopsis

## 1 Introduction

Each year, the Digital Universe – the amount of data stored by humanity – increases by an estimated 40% and in 2013, it was estimated to be 4.4 trillion gigabytes in size [32]. Contributing factors are digital communication, digital media consumption, the rise of the internet of things [15], and public and private digitization efforts. At the same time, the number of people connected to the internet continues to grow: by the end of 2014, an estimated 44% of all households worldwide were connected to the internet and this number is growing at 9% per year [17]. Finally, the amount of available bandwidth world wide is growing with an astounding 45% per year [17]: mankind is generating a wealth of data at ever increasing speeds.

The value of this data deluge is enormous. Some estimate the annual financial value that lies in "Big Data" – data volumes so large that they cannot be processed by conventional computer systems – to be as as much as 300 billion USD for the US health care sector alone [20]. To harness it, technologies that can process large quantities of data in a timely manner are of paramount importance.

No single computer is powerful enough to process, and hence make use of, such large data sets. For this reason, "distributed" system architectures that involve the cooperation of hundreds of low cost "commodity" servers have been proposed [8]. In the last decade, systems built using distributed programming frameworks such as Apache Hadoop MapReduce[2] have been successfully deployed. One major drawback of such systems is that processing is based on large batches of data and that results are only available at the very end of the process. To remedy this, distributed stream processing systems (DSPS) [7] that ingest data and return results continuously have been proposed. This thesis is concerned with studying ways to increase the efficiency of DSPS and with measuring the value of increased processing efficiency.

[2] https://hadoop.apache.org

## 1.1 Background

In this section, the wider context and related work of this thesis are presented. Before introducing different types of DSPS, several approaches and techniques applied in distributed computing are summarized.

**Distributed Computing & Parallelism** To enable services that make use of large amounts of data, processing has to be conducted by multiple machines concurrently. In such applications, processing is "distributed" across a large number of low-cost "commodity" servers that can efficiently operate as compute clusters. While this approach offers large amounts of storage and processing power at an affordable price, new concerns need to be dealt with: as low-cost machines have higher failure rates, robustness against machine failure is a concern. Also, the large number of servers in a cluster incur a heavy load on the network. As these concerns are independent of the underlying application, programming frameworks have been proposed that free application developers from these concerns.

Common to all these frameworks is the goal of parallelizing applications by partitioning and distributing the workload across several machines in a compute cluster. Traditionally, there are two approaches to achieve this and they have been studied in the context of multiprocessor and multicore environments before: In task-parallel systems [19], the application itself is broken into multiple parts and each part is executed on a different resource (i.e. CPU or CPU core). In contrast to this, in data-parallel systems [14], the data that the application operates on is partitioned, and multiple instances of the program operate on multiple parts of the data concurrently. As an example, consider an application that needs to count words over a large corpus of text. The steps involved are cleaning the input text, splitting it into lists of words, and finally, counting the occurrence of each word in the resulting word stream. A data-parallel approach of parallelizing this application would be to partition the input data into multiple parts and to count the words in each part on a separate machine. The results of all partitions is then aggregated in a second step of the process. A task-parallel implementation would be to create a processing pipeline by running each step of the processing as an operator on its machine. The "cleaning" operator would read the data from a source and pass it on to a "splitting" operator. This operator, in

turn, splits the sentences and sends its output – the separated words – to the "counting" operator. Both approaches have different advantages. The data-parallel architecture, for example, can be highly parallelized as one can partition the data into arbitrarily small portions. One disadvantage is that results are only available after all the data has been processed and aggregated. Task-parallel pipelines, in contrast, produce results continuously and iteratively. One disadvantage of a purely task-parallel architecture is that the degree of parallelism is limited by the number of operators in the processing pipeline. As in parallel computing, data-parallelism and task-parallelism can be combined to make use of both their respective advantages [30]. In fact, many distributed computing frameworks combine these two forms of parallelism and in some cases, allow the user to choose the degree of task- and data-parallelism through configuration options. For this reason, it is often difficult to uniquely classify a system as being either data-parallel or task-parallel. However, most programming frameworks have one of the two parallelism approaches at their core and accommodate the respective other form of parallelism through extensions.

The first widely used distributed programming frameworks were data-parallel – also called "batch-based" – systems. In 2004, the two Google employees, Jeffrey Dean and Sanjay Ghemawat, presented their *MapReduce* framework, which applied the idea of data-parallelism to distributed computing [8]. MapReduce enabled software engineers within Google, to run applications on large compute clusters, without having to worry about the complexities of distributed computing, by simply developing their applications within this framework. A MapReduce program consists of a mapper function – a function that computes a key-value for each data item – and a reducer function – a function that combines several data items that share the same key-value. Whenever possible, instead of sending data across the network, the computer program is sent to each storage machine and executed over the part of the data that resides on the respective machine (map phase). This approach is inspired by an idea called "active disks" [24], in which computation is moved away from the main CPU into the peripherals (the disks). Engineers at Yahoo implemented these ideas in the open source Apache project Hadoop (Hadoop). Google did not release the source code for their MapReduce system. Possibly due to this, Hadoop became the de-facto industry standard for Big

Data processing, and many modern distributed computing frameworks operate on or are compatible with Hadoop.

The MapReduce paradigm is ideally suited for processing needs that can be expressed as a series of mapper and reducer steps. Several projects extend Hadoop MapReduce (HMR) by providing means to concatenate several HMR jobs into a processing pipeline (Apache Oozie[3]) or generate HMR pipelines on the fly by compiling a program specified in a separate language into a series of HMR jobs (Apache Pig[4]). However, one disadvantage of HMR-based applications is that each HMR job needs to fully finish before the next job can start. In other words, in a processing pipeline of multiple HMR jobs (tasks) there is always only one task being executed at any time. This is problematic for applications that require short, iterative, or continuous processing. For example, in scenarios where streams of data need to be continuously ingested and results are required to be available in a timely manner, HMR is an ill-suited solution. For this reason, several teams proposed task-parallel frameworks that are geared towards distributed realtime processing of data streams.

**Distributed Stream Processing** To reduce reaction times of distributed systems, distributed task-parallel-centric – also called "stream-based" – programming approaches have been proposed. These systems are designed around the idea of an operator graph in which data is passed from operator to operator. One early representative of such a system is Borealis [2], which is an extension of the Aurora stream processing engine [1] and the Medusa distributed message passing system [5]. In Borealis, applications are specified as a graph of operators (processing boxes) and connections between them (arrows). The application is executed on a compute cluster, and the workload is distributed across the compute cluster on a per-operator basis. A similar system is IBM's Stream Processing Core (SPC) [4]. These early systems implemented task-parallelism, but they did not offer any form of data-parallelism, and hence limited the degree of parallelism of applications to the number operators in the program. Early studies of task-parallel systems that included (partial) data-parallelism were Flux [27] and Aurora* [7]. Both systems concentrated on data-parallelism on a per-operator level and were either of conceptual

[3] http://oozie.apache.org
[4] http://pig.apache.org

nature [7] or only implemented as a simulation [27]. The first programming framework for task-parallel applications with end-to-end support for data-parallelism was Microsoft's Dryad system [16]. A Dryad application is a directed graph of operators that are connected by channels. To support data-parallelism, and inspired by Google's MapReduce [8], the system is capable of reading partitioned input and of processing it through a concatenation of several replicated "virtual vertices". Dryad was open-sourced in 2009, two years after its initial publication in 2007. Since 2009, many more companies published research about their task- and data-parallel realtime processing systems such as IBM's System S (successor of SPC) [26], Yahoo's S4 [22], Twitter's Storm,[5] Google's Millwheel [3], and the two Microsoft systems Naiad [21] and Timestream [23]. In all of these systems, applications are directed graphs of operators that communicate messages downstream.

While this thesis is concerned with the study of DSPSs consisting of task-parallel-centric operator graphs, it is worth mentioning that there are also distributed stream processing frameworks that are data-parallel at the core: The Apache Spark project,[6] in essence, is an in-memory based version of the MapReduce concept, although it offers many more operators than just mappers and reducers. The basic building block, it operates on, is the Resilient Distributed Dataset (RDD). Spark has a streaming component, which offers data-parallel in-memory processing of small batches of data (RDDs) to accommodate streaming use cases [34].

As there are already a large number of DSPS available, the goal of this thesis is not to propose yet another system, but to investigate, on the one hand, how to make DSPS more efficient, and, on the other hand, to explore the benefits of more timely Big Data processing through DSPS in terms of improved end-user experience. A concrete definition of the problems tackled in this thesis is given in the next section.

## 1.2 Problem Statement

Much of the existing research in DSPS has focused on programming paradigms [16, 22], query languages [11, 13, 21, 23], or resilience against

[5] https://storm.incubator.apache.org
[6] https://spark.apache.org

machine failure [3, 22]. Less work has been conducted in the area of making existing frameworks more efficient. Areas of interest so far have been load-elasticity [26, 12] and auto-parallelization [33]. Fewer teams have tackled the field of general auto-configuration [35, 25]. For this reason, the work in this thesis is concerned with investigating techniques to improve the performance of DSPSs through configuration.

Distributed applications rely on the interplay of many computers. This not only increases the complexity of the overall system, but also makes predictions about its runtime behavior more difficult. To put it another way, configuring a distributed system is a non-trivial task. Further, when designing DSPSs, an additional challenge is the fact that the data to be processed is unknown ex-ante, and that the system may need to adapt to changing circumstances. Thus, some performance improvements of a DSPS necessarily have to be conducted based on runtime information of the system and cannot be done in the configuration or deployment phase.

It seems intuitive that higher performance and lower latency of a stream processing system should lead to a higher quality services for end users. However, as cluster resources and programming frameworks geared towards distributed stream processing have only become available in recent years, few research has been published on real-world deployments of such systems.

To summarize: there exist many frameworks to build DSPS that are non-trivial to configure. As these framework are relatively new, little research on real-world deployments of DSPS exist. Hence, this thesis is concerned with the following aspects of DSPS:

1. How to increase the efficiency of DSPS?
2. What is the value of increased efficiency of a DSPS?

The approach taken to investigate these issues is described next.

## 1.3 Purpose and Nature of this Thesis

To investigate the issues outlined in section 1.2, we built real-world DSPS and conducted experiments using several different performance optimization techniques. We measured system performance and report on our empirical findings. We compare our results against the performance of approaches proposed by other research teams and as well as against approaches used in industry. As we are working with distributed

stream processing systems, we are mainly interested in lowering network load in the cluster and in increasing throughput, and hence reducing latency of the overall system. To assess the service quality improvements that result from faster data processing, we also measured qualities such as relevance and freshness of a service in a user study.

We present concrete research questions along with hypotheses about how to tackle them in the next section. As this is a cumulative dissertation consisting of three work packages, each question outlined below is treated in a separate research project. Summaries of all three work packages are presented in Section 2, and the full papers are attached in Part II.

## 1.4 Research Questions & Hypotheses

In this section, the research questions addressed in this thesis are presented and hypotheses stated accordingly. Each will briefly be motivated.

**Increasing Efficiency of DSPS Using Runtime Information** As distributed systems consist of many computers that communicate over a network, one limiting factor for the scalability of such systems is the required communication capacity (bandwidth) between cluster nodes, for the system to function. In large data centers, bandwidth has historically been a scarce resource [8]. Large internet companies such as Facebook are experiencing up to three orders of magnitude more network traffic within their data centers than the traffic with the outside world [9]. For this reason, reducing bandwidth requirements of a DSPS is desirable, as it increases the overall amount of processing that can be conducted in a data center. For task-parallel systems, data sent between operators (tasks) consumes bandwidth if the communicating operators are not placed on the same physical machine. For task-parallel systems, inter-operator communication is dependent on the data that is being processed. In stream processing systems, the data to be processed is unknown ex-ante. Thus, the communication behavior of the system is only apparent at runtime. For this reason, the first research question that we tackle in this thesis is how to reduce the amount of required bandwidth of a DSPS using runtime information of the system itself.

One way to reduce the amount of communication between parts of a processing graph is to co-locate components that exhibit high communi-

cation load between. This process is called workload scheduling. Given that many of approaches to build DSPS are based on data-parallel processing graphs (see Section 1.1), it seems intuitive to try and use graph-based optimization techniques to achieve this. Graph partitioning algorithms promise to partition graphs so that strongly connected sub-groups are assigned to the same partition. This aspect seems like a perfect fit for the problem at hand, so the first hypothesis of this thesis is:

**HYPOTHESIS 1**: Task-parallel DSPS can effectively be scheduled by applying graph partitioning algorithms on system runtime information.

**Increasing Efficiency of DSPS using Auto-Configuration** Distributed systems often offer many configuration parameters that allow users to tune aspects of the system to increase overall performance [25]. The performance of the distributed system is partially dependent on the value of each parameter setting and its interplay with all the other settings. Other important factors are the idiosyncrasies of the application itself, the data that it processes, and of the underlying hardware the system is running on. As applications become more complex, it becomes ever more difficult to predict the effect that the value of each parameter setting has on the overall system. The performance of the system can hence be modeled as a black-box function that needs to be optimized. Thus, a second research question that we explore in this thesis, is what suitable optimization techniques to apply to a DSPS to find good configuration settings to improve the performance.

One approach which is suitable for maximizing black-box functions is the technique of Bayesian Optimization. This technique has successfully been applied in other fields such as machine learning [6, 31, 28] and it seems intuitive to try and apply this technique to the problem of DSPS configuration. Hence, the second hypothesis of this thesis is:

**HYPOTHESIS 2**: Performance of DSPS can be increased through auto-configuration using black-box optimization strategies, such as Bayesian Optimization.

**Assessing the Impact of Increased Service Timeliness** It is well known that reducing latency of services has economic value. Observations by companies such as Amazon, Google, and Yahoo of reduced number of

sales, reduced traffic, and reduced number of searches that correlate with increased latency,[7] or the recent surge in high frequency trading [29] are a testament to this. The backends that are used to create these services, however, have only recently started to be upgraded with timeliness in mind. As more distributed backend systems are being re-engineered to make use of frameworks that are geared towards realtime processing, the question of the value of this increased timeliness arises. Intuitively, it makes sense to assume that reduced latency and increased timeliness would yield increased relevance of the resulting products or services. Hence, the final hypothesis of this thesis is:

**HYPOTHESIS 3**: The quality of a service can be increased by reducing computation latency resulting from performance improvements of the underlying processing pipeline.

The next chapter gives a summary of how the presented questions were answered and the posed hypothesis were tested in this thesis.

## 2 Contribution Summaries

This section gives a short summary of the individual research projects that this thesis comprises of. Each project is motivated, succinctly summarized, and its findings are put into relation to the hypotheses proposed in Section 1.4.

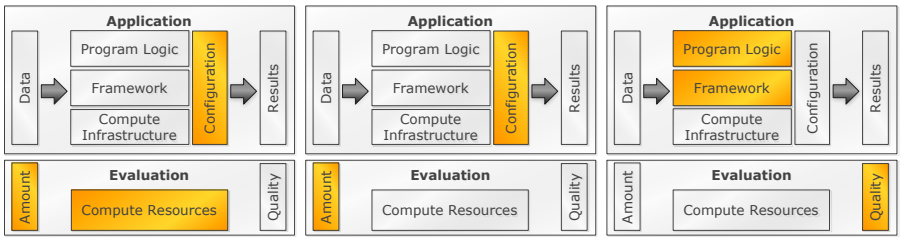
To support the understanding of the differences and similarities between the three projects, Figure 1 graphically depicts the focus of each project: All three projects were concerned with implementing and evaluating applications of DSPS. Such applications typically consist of a program logic implemented using a programming framework running on some compute infrastructure (the cluster). The goal of these applications is to process some amount of input data to arrive at quality results while using as little of the available compute resources as possible. Configuration is chosen so that the application as a whole operates efficiently.

The applications used in this thesis were evaluated along several dimensions: With all projects, the goal was to increase the efficiency and/or the effectiveness of the overall system. The former, efficiency, can be measured through computational metrics such as the amount of data that

[7] http://www.slideshare.net/stubbornella/designing-fast-websites-presentation

can be processed in a given unit of time (throughput) or the amount of required compute resources (load) to process the input data. The latter, effectiveness, can be measured via, for example, a quality metric of the computed results.

In the first two projects, depicted in Figure 1a and 1b, the focus lay on improving the efficiency of DSPSs through configuration. Improved efficiency was achieved by increasing the throughput of the system and/or by reducing the required network bandwidth (compute resource) through intelligent and automated configuration strategies and workload scheduling. The last project, depicted in Figure 1c, was concerned with evaluating the increase in effectiveness that resulted from increased processing efficiency of a DSPS. More detailed descriptions of these projects follow in the sections below.




(a) Workload Scheduling (b) Bayesian Optimization (c) Timely Semantics

**Fig. 1.** The focus in each of the three research projects of this thesis.

## 2.1 Online Workload Scheduling in Distributed Stream Processors using Graph Partitioning

In the first project of this thesis, depicted in Figure 1a, we explored a way to reduce the network load of a distributed stream processor. For this project, we concentrated on DSPS that are based on task- and data-parallel operator graphs – *topologies*. The goal was to organize the worker nodes of the topology – the *task instances* – in a way that leads to reduced bandwidth utilization while retaining or increasing throughput. This process is called workload scheduling.

To this end, we employed a graph partitioning algorithm to distribute task instances among available servers in a compute cluster. The graph to

be partitioned was first built by collecting the communication behavior of task instances of the running system. In a second step, a schedule was computed using a graph partitioning algorithm and applied to the running system. We empirically measured the effect of this scheduling strategy using four different use cases on cluster configurations of between 10 and 80 machines.

We found that this scheduling strategy not only reduces bandwidth consumption significantly (by up to 88%), but, depending on the size of the payload being sent around in the topology, can also significantly increases the throughput of the overall system (by up to 56%). We compared the effectiveness of our graph partitioning based scheduler against two other schedulers, both of which performed worse than our approach. These findings support Hypothesis 1, which suggests that graph partitioning can be used to effectively schedule task-parallel DSPS.

## 2.2 Machines Tuning Machines: Configuring Distributed Stream Processors with Bayesian Optimization

The second project, depicted in Figure 1b, is concerned with an automated solution for tuning a DSPS using the technique of Bayesian Optimization. When developing a DSPS, there are many configuration options that need to be adjusted to the specifics of an application deployment, and the performance of the overall system is partially dependent on these parameter settings. However, given the complexity of modern DSPS, the impact of every single parameter setting is hard to predict and the interaction between all parts of a DSPS is often impossible to foresee. Thus, the whole system can be seen as a black-box. In this project we explored the possibility of automatically configuring a DSPS using a black-box optimization method called Bayesian Optimization.

We implemented a series of synthetic as well as one real-world DSPS application and had a Bayesian Optimization framework find good parameter settings for these applications. We compared the throughput performance of the resulting systems with a baseline obtained from configuring the same pipelines using a parallel linear approach.

We found that, given enough time, Bayesian Optimization is a very viable tool for configuring complex DSPSs. Bayesian Optimization allowed us to increase the performance of the DSPSs in our testbed by a factor of up to 2.8 in the best case. For some aspects of a DSPS, as for

example auto-parallelization, other approaches, such as using topological information, seem to be more effective than Bayesian Optimization. However, in cases where topological information is unavailable, or expensive to obtain, Bayesian Optimization is a viable alternative. In conclusion, these results support Hypothesis 2, which suggests that Bayesian Optimization can be leveraged to increase the performance of a DSPS.

## 2.3 Timely Semantics: A Study of a Stream-based Ranking System for Entity Relationships

In the third research project, depicted in Figure 1c, we investigated the impact of a more timely processing pipeline on the quality of a user-facing service. To this end, we re-implemented a part of a MapReduce-based entity ranking pipeline in production at Yahoo, using Apache Storm – a programming framework that is geared towards distributed realtime processing. Both ranking systems rank related entities through a learning-to-rank approach based on user generated information such as search log data. Re-implementing this pipeline using a streaming-based framework enabled us to compute entity rankings in less time compared to the MapReduce-based system, which in turn allowed us to generate entity rankings using more recent search log data.

We evaluated the impact of being able to process more recent data on the quality of the rankings in a longitudinal user study: We generated entity rankings using nine different datasets of three different sizes and three different age categories. We then had professional human judges evaluate the quality of the resulting rankings over the course of four different days and quantitatively analyzed the results.

We found that processing more up-to-date data not only increased the freshness of the computed rankings, but also increased overall relevance. We statistically showed, that increased freshness is a predictor for higher relevance. Finally, we found that in some of our experiments, the use of more recent search log data could even compensate for cases where the amount of available search log data was limited. This is important, as the entity-ranking pipeline is based on a decision tree model, and hence, the amount of data available during the training phase impacts the quality of the computed rankings. Thus, Hypothesis 3 of this thesis, which is concerned with the question whether increased processing speeds can be leveraged to achieve increased service quality, is supported by these

results. This is, if we consider the two qualities of freshness and relevance of entity rankings produced from search log data.

# 3 Limitations

In this section, common limitations of the projects presented are discussed. More details on limitations of the each project can be found in the respective sections in Part II of this thesis.

**Too Many Knobs** In order for research results to be meaningful, experiments need to at least partially replicate real-world situations. On the other hand, experiments need to be designed in a way that allow the researcher to focus on separate aspects of a system. The reason why programming frameworks have been proposed to build distributed systems is that such systems are inherently very complex. Frameworks like Apache Storm hide many of the complexities of distributed systems and allow the user (the programmer) to adapt the system to the specifics of the underlying hardware and software through configuration settings. This leads to the situation in which the overall system is dependent on so many factors, that it becomes hard to arrive at generalizable results. However, not running experiments on real infrastructure and only simulating a cluster, can lead to results that not accurately reflect the real system. For example, before implementing the project presented in Section 2.1, we simulated the effect of the scheduler in a preliminary evaluation [10]. The results we measured in the simulation were much more promising than what we eventually observed when we evaluated the deployment on a real cluster. The reason for this was, that we overlooked certain aspects of the real-world implementation, such as certain aspects of Storm's bookkeeping facility.

This dilemma is getting worse as industry deployments become more complex. This, in turn, leads to another limitation that can be observed in distributed systems research: The deployments in university labs are often very different from what is used in industry.

**Lab vs. Industry** As backend systems often comprise the very heart of the competitive advantage of companies, these companies are naturally secretive about them. Real-world deployments are hard to come by, so

in many cases, research in university labs necessarily is conducted on synthetic applications. Synthetic problems, however, can only partially simulate real deployments. The problem here is the same as in the previous section: The interplay of all components within distributed systems is often so complex that simulations are bound to leave out factors. After all, this complexity is the reason why distributed programming frameworks have been proposed in the first place. The consequence of this is that the experiments in the research projects presented in this thesis may not accurately reflect the situation of real-world deployments.

# 4 Conclusions & Future Work

The goal of the work presented in this thesis was to investigate strategies that allow us to cope with the ever increasing amount of data we generate. To this end, we implemented and evaluated DSPS applications and empirically evaluated various of their aspects in three separate studies: on the one hand, we looked at strategies to make distributed stream processing more efficient, and on the other hand, we explored the effectiveness improvements that can be achieved through more efficient distributed stream processing. We showed how existing algorithms can successfully be applied to the domain of DSPS to increase the performance of these systems, and we showed one way in which such increased performance can lead to higher levels of quality (higher effectiveness) of user-facing services.

In the first study, we implemented and evaluated a workload scheduling strategy for task-parallel DSPS based on graph-partitioning. Using several real-world and synthetic use cases, we showed that this approach can lead to significant reductions in bandwidth consumption and increased throughput compared to two competing scheduling algorithms. This line of work could be extended in several ways. For instance, while our current implementation uses simple message counts to approximate CPU and network loads, a more fine grained collection of system information may be fruitful. Further, an implementation that collects multiple performance metrics and applies multi-constraint graph partitioning [18] may yield even better results.

In the second study, we applied Bayesian Optimization to the task of automatically configuring DSPSs. We, again, evaluated our approach using several synthetic use cases as well as one real-world use case. We

compared the Bayesian Optimization approach to a naive parallel as well as to an informed parallel approach. We showed that our black-box approach can partially compensate for missing topological information in auto-parallelization. Further, our approach led to increased throughput values of a factor of up to 2.8 through auto-configuration of system settings. While these results are promising, more research will be necessary to evaluate the usefulness of Bayesian Optimization in the realm of DSPS. Next to evaluating the approach using a larger set of real-world applications, it would also be interesting to compare Bayesian Optimization against other auto-configuration techniques such as for example Covariance Matrix Adaption (CMA) [25]. Last, our auto-configuration approach using Bayesian Optimization could potentially be useful in other settings: many modern programming frameworks for distributed systems offer a large number of configuration options and there is no apparent reason as to why the approach we have taken to configure our DSPS could not be applied to other distributed systems as well.

For the last study, we measured the increase in ranking quality of an entity-ranking system that resulted from increased processing speed and hence data recency. In a longitudinal user study we compared the ranking quality generated using varying amounts and ages of search log data and showed, that the ability to process more recent data not only increases the freshness, but also the overall relevance of the generated rankings. We also found, that recency of data can be more important to ranking quality than data volume: in some cases, fresher and more relevant rankings resulted from processing less, but more recent search log data. This project could be extended as follows: In this study we only evaluated the effect of using more recent input data to compute rankings. We did not make use of the fact that a continuous streaming system also offers the possibility to create new features to train the decision tree models on. For example, instead of only looking at static counter values, using features that capture change over time (trends) may yield more relevant results.

Concluding from the review of related work as well as current developments in computing and information processing, the trend towards timely and large-scale data processing is likely to continue. Given the limits of modern computer systems, it is further safe to assume that much of this processing will happen in a distributed setting. We therefore dare to hope that the work presented in this thesis will be of continued value

to the community of researchers and practitioners in distributed stream processing.

# Bibliography


[1] D. Abadi, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, S. Zdonik, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, and A. Maskey. Aurora: a data stream management system. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*. Association for Computing Machinery (ACM), 2003. doi: 10.1145/872757.872855. URL `http://dx.doi.org/10.1145/872757.872855`.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2005 CIDR Conference*. Citeseer, Citeseer, 2005. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.7039&rep=rep1&type=pdf`.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Whittle Sam. Millwheel : Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6 (11), 2013. URL `http://research.google.com/pubs/archive/41378.pdf`.

[4] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *In Proceedings of the Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, 2006.

[5] Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - NSDI '04*. USENIX Association, 2004. URL `http://dl.acm.org/citation.cfm?id=1251175.1251190`.

[6] James Bergstra, D Yamins, and D Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *Proceedings of the 30th International Conference on Machine Learning*, 2013. URL http://jmlr.org/proceedings/papers/v28/bergstra13.html.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *First Biennial Conference on Innovative Data Systems Research - CIDR '03*, 2003.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - OSDI '04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[9] Nathan Farrington and Alexey Andreyev. Facebook's data center network architecture. In *2013 Optical Interconnects Conference*. Institute of Electrical & Electronics Engineers (IEEE), may 2013. doi: 10.1109/oic.2013.6552917. URL http://dx.doi.org/10.1109/OIC.2013.6552917.

[10] Lorenz Fischer, Thomas Scharrenbach, and Abraham Bernstein. Scalable linked data stream processing via network-aware workload scheduling. In *International Workshop on Scalable Semantic Web Knowledge Base Systems - SSWS '13*, 2013. URL http://ceur-ws.org/Vol-1046/.

[11] Buğra Gedik, Henrique Andrade, and Kun-Lung Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*. Association for Computing Machinery (ACM), 2009. doi: 10.1145/1645953.1646061. URL http://dx.doi.org/10.1145/1645953.1646061.

[12] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, jun 2014. doi: 10.1109/tpds.2013.295. URL http://dx.doi.org/10.1109/TPDS.2013.295.

[13] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, dec 2012. doi: 10.1109/tpds.2012.24. URL http://dx.doi.org/10.1109/TPDS.2012.24.

[14] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, dec 1986. doi: 10.1145/7902.7903. URL http://dx.doi.org/10.1145/7902.7903.

[15] Antonio Iera, Christian Floerkemeier, Jin Mitsugi, and Giacomo Morabito. The internet of things [guest editorial. *IEEE Wireless Commun.*, 17(6):8–9, dec 2010. doi: 10.1109/mwc.2010.5675772. URL http://dx.doi.org/10.1109/MWC.2010.5675772.

[16] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 - EuroSys '07*. Association for Computing Machinery (ACM), 2007. doi: 10.1145/1272996.1273005. URL http://dx.doi.org/10.1145/1272996.1273005.

[17] ITU. Measuring the information society report. Technical Report 6, International Telecommunication Union (ITU), 2014.

[18] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the IEEE/ACM SC98 Conference*. Institute of Electrical & Electronics Engineers (IEEE), 1998. doi: 10.1109/sc.1998.10018. URL http://dx.doi.org/10.1109/SC.1998.10018.

[19] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, jan 1987. doi: 10.1109/tc.1987.5009446. URL http://dx.doi.org/10.1109/TC.1987.5009446.

[20] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, may 2011.

[21] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2517349.2522738. URL http://dx.doi.org/10.1145/2517349.2522738.

[22] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*. Institute of Electrical & Electronics Engineers (IEEE), dec 2010. doi: 10.1109/icdmw.2010.172. URL http://dx.doi.org/10.1109/ICDMW.2010.172.

[23] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2465351.2465353. URL http://dx.doi.org/10.1145/2465351.2465353.

[24] Erik Riedel, Christos Faloutos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, jun 2001. doi: 10.1109/2.928624. URL http://dx.doi.org/10.1109/2.928624.

[25] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *The 28th International Conference on Distributed Computing Systems*. Institute of Electrical & Electronics Engineers (IEEE), jun 2008. doi: 10.1109/icdcs.2008.11. URL http://dx.doi.org/10.1109/ICDCS.2008.11.

[26] Scott Schneider, Henrique Andrade, Buğra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. Institute of Electrical & Electronics Engineers (IEEE), may 2009. doi: 10.1109/ipdps.2009.5161036. URL http://dx.doi.org/10.1109/IPDPS.2009.5161036.

[27] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: an adaptive partitioning operator for

continuous query systems. In *Proceedings 19th International Conference on Data Engineering*. Institute of Electrical & Electronics Engineers (IEEE), 2003. doi: 10.1109/icde.2003.1260779. URL http://dx.doi.org/10.1109/ICDE.2003.1260779.

[28] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems - NIPS '12*, 2012. URL http://arxiv.org/abs/1206.2944.

[29] Hans R. Stoll. High speed equities trading: 1993-2012. *Asia-Pacific Journal of Financial Studies*, 43(6):767–797, dec 2014. doi: 10.1111/ajfs.12078. URL http://dx.doi.org/10.1111/ajfs.12078.

[30] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. *ACM SIGPLAN Notices*, 28(7):13–22, jul 1993. doi: 10.1145/173284.155334. URL http://dx.doi.org/10.1145/173284.155334.

[31] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2487575.2487629. URL http://dx.doi.org/10.1145/2487575.2487629.

[32] Vernon Turner, David Reinsel, John F. Gantz, and Stephen Minton. The digital universe of opportunities: Rich data and increasing value of the internet of things. Technical Report April, IDC, 2014.

[33] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems - DEBS '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2335484.2335515. URL http://dx.doi.org/10.1145/2335484.2335515.

[34] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/

2517349.2522737. URL http://dx.doi.org/10.1145/2517349.2522737.

[35] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219, jun 2007. doi: 10.1145/1272998.1273020. URL http://dx.doi.org/10.1145/1272998.1273020.

# Part II

# Papers

# Online Workload Scheduling
## in Distributed Stream Processors using Graph Partitioning

This chapter is based on a paper that has been accepted at and will be published in the proceedings of the *2015 IEEE International Conference on Big Data (IEEE BigData 2015)*. It is an extension of the work that we published at the *9th International Workshop on Scalable Semantic Web Knowledge Base Systems* which was co-located with the *12th International Semantic Web Conference (ISWC 2013)* [9].

# Online Workload Scheduling
## in Distributed Stream Processors using Graph Partitioning

Lorenz Fischer and Abraham Bernstein

[1] DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
lfischer@ifi.uzh.ch
[2] bernstein@ifi.uzh.ch

**Abstract.** With ever increasing data volumes, large compute clusters that process data in a distributed manner have become prevalent in industry. For distributed stream processing platforms (such as Storm) the question of how to distribute workload to available machines, has important implications for the overall performance of the system.
We present a workload scheduling strategy that is based on a graph partitioning algorithm. The scheduler is application agnostic: it collects the communication behavior of running applications and creates the schedules by partitioning the resulting communication graph using the METIS graph partitioning software. As we build upon graph partitioning algorithms that have been shown to scale to very large graphs, our approach can cope with topologies with millions of tasks. While the experiments in this paper assume static data loads, our approach could also be used in a dynamic setting.
We implemented our proposed algorithm for the Storm stream processing system and evaluated it on a commodity cluster with up to 80 machines. The evaluation was conducted on four different use cases – three using synthetic data loads and one application that processes real data.
We compared our algorithm against two state-of-the-art scheduler implementations and show that our approach offers significant improvements in terms of resource utilization, enabling higher throughput at reduced network loads. We show that these improvements can be achieved while maintaining a balanced workload in terms of CPU usage and bandwidth consumption across the cluster. We also found that the performance advantage increases with message size, providing an important insight for stream-processing approaches based on micro-batching.

## 1 Introduction

When work has to be distributed across a compute cluster, scheduling—the process of deciding which cluster machine is assigned which part of the workload—is of paramount importance for the overall performance of the system. Extensive research has been conducted in the realm of scheduling work in data-parallel systems. In data-parallel systems, data is partitioned into small units, which are then assigned to worker nodes in a compute cluster to process. A prominent representative of data-parallel

systems is Apache Hadoop[3] which is the most popular implementation of MapReduce [8]. Rao and Reddy [22] give an overview over several scheduling strategies within the Hadoop MapReduce framework such as FiFo, fair, capacity, delay, deadline, and resource-aware schedulers.

In contrast to data-parallel systems, task-parallel applications are designed as a set of tasks that run in parallel on a cluster for indefinite time. While these systems also incorporate data-parallelism, the processing is divided across the cluster and the data is partitioned and routed to task instances, accordingly. Google's Millwheel [2], Microsoft's Naiad [18] and Timestream [21], IBM's Infosphere Streams [23], as well as Apache Storm[4] are representatives of such systems. Workload schedulers of task-parallel systems need to distribute the compute tasks in a way that makes optimal use of the available compute resources.

In this study, we present a workload scheduler for task-based distributed stream processing systems that is based on a graph partitioning algorithm. We implemented the scheduler for the Apache Storm platform and measured its performance in an extensive empirical evaluation. This work is a continuation of ideas presented at a workshop where we first proposed building a Storm scheduler based on graph partitioning [9]. While this previous work was based on simulations, the study at hand presents an evaluation on actual compute clusters in a real world setup. In particular the contributions of this study are as follows:

1. We present an implementation of a scheduling algorithm to schedule operators in a distributed task-parallel stream processing system which is based on graph partitioning.
2. We present a set of benchmark topologies with their associated data to evaluate our scheduler on the Storm realtime processing framework.
3. We evaluate our algorithm against two alternative approaches in an extensive evaluation on a wide range of varying cluster configurations.
4. We report on key lessons learned and discuss the implications of our observations for the field of distributed stream processing.

The remainder of this paper is organized as follows: We give an overview of related work in Section 2, before introducing our algorithm in Section 3. We then present the experimental setup and our results in

[3] http://hadoop.apache.org
[4] https://storm.incubator.apache.org

Sections 4 and 5, respectively. We close with a discussion of our findings in Section 5.

## 2 Related Work

Our work is related to the fields of workload scheduling in distributed systems as well as using graph partitioning algorithms for resource allocation. Here we succinctly report on related work in these fields and discuss our research in its light.

### 2.1 Workload Scheduling in Distributed Systems

Most research in workload scheduling for distributed systems has been conducted for batch-based systems. In such systems data is partitioned into small shards that are then assigned to worker nodes. [22] reviews common scheduling strategies for data-parallel systems. We only mention Sparrow [19] here, because its application is a low latency data-parallel system that could be used in a streaming setup. Sparrow is a low latency task scheduler for data-parallel systems based on a decentralized randomized sampling approach that has been evaluated in the Spark [27] environment. It continuously assigns tasks of processing small batches of data based on local information to minimize the delay in job execution. The overarching goal of this work is to achieve low latency in scheduling. Our study focuses on task-parallel processing [23], where the overall system performance in terms of throughput and network utilization takes precedence over the latency of scheduling a single computation.

In contrast to batch-based distributed systems, where scheduling speed can become an issue, in a task-based environments, workload assignments can persist for longer durations. One early representative of a distributed task-based stream processing system is Borealis [1], which places operators of a streaming system across geographically distributed computers. Pietzuch et al. presented their scheduler that is based on a *stream-based overlay network* (SBON) optimizing operator placement [20]. Xing et al. presented two different scheduling approaches for the Borealis System. In [25] they propose a greedy heuristic to find an optimal operator placement in polynomial time and in [26] they propose to find an operator placement that is "resilient" to change, meaning that it does not have to be changed upon load changes. Heinze et al. model the

problem of operator placement in Borealis as a bin-packing problem and use a firstfit heuristic to assign operators to machines (bins) [12].

Other task-based systems distribute work across a set of computers within the same data center. For example, Isard et al. presented a scheduling system for Microsoft's Dryad [13] in [14]. This scheduler maps the problem of task to worker assignment into a graph over which a min-cost flow algorithm then minimizes the cost of a model which includes data locality, fairness, and starvation-freedom. Wolf et al. presented the scheduling system SODA [24], which is the workload scheduling system of System S (later renamed to IBM Infosphere Streams). In SODA, the assignment of processing elements (PE) to cluster nodes is usually handled by a mixed integer program implemented in CPLEX[5] to balance CPU and bandwidth constraints. When the mixed integer program fails, due to the quadratic complexity of the problem, then either round-robin or a heuristics-based scheduling mechanism is used. The former essentially corresponds to Storm's default scheduling strategy. The latter tries to assigns all communicating PE pairs to the same machine incrementally starting from the pair that communicates most.

Aniello et al. [5] present two scheduling algorithms for Storm. Their *offline* scheduler bases its work assignments on an analysis of the topology of the program, their *online* scheduler takes runtime CPU and network load characteristics of the cluster into account following a procedure conceptually similar to heuristic module employed by SODA: all communicating processing pairs of Storm tasks (Storm's equivalent of a PE) are sorted according to the amount of communication between them and assigned to worker nodes on a best-effort basis.

In Naiad [18], one shard of each operator is assigned to each worker[6], a strategy that for certain configurations is equivalent to the default scheduler of Storm.

Finally, two query planning systems need to be mentioned: In StreamCloud [11], information contained in the query is leveraged to do the scheduling. Hence, each operator that is used in the query needs to have a counter part in the scheduler. As inter-operator communication is handled in the query compiler, StreamCloud's greedy load balancing (scheduling) algorithm focuses on machine load in terms of CPU. Kaly-

[5] http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer
[6] https://bigdataatsvc.wordpress.com/2012/10/18/running-distributed-naiad-programs

vianaki et al. present an approach that solves the $NP$-hard multi-query planning problem by cleverly approximating an optimal placement [15]: in their SQPR planner, they incrementally add queries to the cluster, trying to re-use parts of existing queries where possible. In contrast to these works, the approach taken by Storm is to separate the task of scheduling from the query compiler, which allows arbitrary code to run in the operators.

Our approach differs from the above-mentioned schedulers in that we employ a graph partitioning algorithm of the METIS software package [16]. In contrast to a min-flow approach, we consider both network and machine load. Our approach differs from SODA's mixed integer program in that we use graph partitioning algorithms that have been shown to scale to millions of edges, using heuristics to avoid the complexity constraints [16]. Hence, our investigation will compare to the round-robin base-line and Aniello's on-line algorithm that is conceptually similar to the latter of SODA's heuristics.

## 2.2 Graph Partitioning for Scheduling

Others have employed graph partitioning algorithms to the problem of workload scheduling before us: Aletá et al. [4] use graph partitioning algorithms to assign instructions to different clusters inside a microprocessor. Similar to our use-case, their goal is to group instructions/operations into clusters in order to balance the workload whilst minimizing inter-cluster communication. Curino et al. use METIS for database replication [7]: They map tuples in the database as nodes and transactions as edges of a graph and let METIS figure out an optimal replication strategy. [6] gives a survey of other applications including "VLSI circuit layout, image processing, solving sparse linear systems, computing fill-reducing orderings for sparse matrices, and distributing workloads for parallel computations." Lastly, and most closely related to our application, graph partitioning has even been applied in stream processing in System S. Their *offline* compiler uses a graph partitioner to fuse multiple processing elements (PEs) of a stream processing graph into bigger PEs in order to decrease inter-process communication [17]. In contrast, *we use information that we collected in an online* phase to inform just-in-time re-scheduling of the workload in the running system. Hence, we

can exploit actual run-time information such as actual computational effort and network usage to optimize our schedule.

# 3 Scheduling Algorithm

In this section, we will describe our approach. We will first give more details about Storm. Then we formally describe how we map the problem of workload scheduling into a graph partitioning problem.

## 3.1 Distributed Stream Processing with Storm

In contrast to batch-based distributed systems such as Apache MapReduce,[7] Storm[8] ingests data continuously. As in MapReduce, a Storm application allows the user to partition the data and to distribute parts of the processing across a compute cluster.

A Storm application—a *topology*—is a directed graph consisting of spout and bolt nodes as the one depicted in Figure 1 on the left. Spouts emit data and bolts consume data from upstream nodes and emit data to downstream nodes. Spout nodes are typically used to connect a Storm topology to external data sources such as queues, web-services, or file systems. For each spout and bolt, the programmer defines how many instances of this node should be created in the physical instantiation of the topology – the task instances. These task instances, or *tasks*, are distributed across all machines of the compute cluster, to which a topology has been assigned. Each edge in the topology graph defines a *grouping strategy* according to which messages that pass between the nodes—the *tuples*—are sent to downstream nodes. This results in a physical topology depicted on the right of Figure 1, which is different than the logical representation.

Tuples are lists of key-value pairs. The programmer defines the tuple format for each edge of the topology (e.g. field1=query_terms, field2=browser_cookie, field3=timestamp). Different grouping strategies provide different guarantees. For example, the field grouping strategy guarantees, that all tuples that share the same value in one or multiple configurable fields are sent to the same task instance. This can be essential for the correctness of the topology: consider the case in which a bolt

[7] http://hadoop.apache.org
[8] https://storm.apache.org

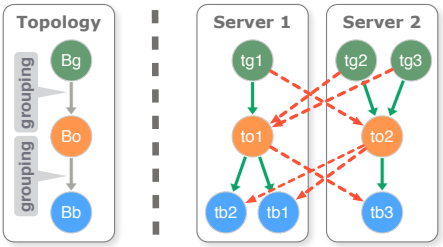



**Fig. 1.** Logical (left) and physical (right) representation of a topology.

is to compute an aggregate income for each age group of a population. In order to compute these aggregate values, we have to ensure that all tuples that share the same age group are sent to the same task instance of the bolt that computes the average. Other grouping strategies may fulfill different purposes. The shuffle grouping, for example, evenly distributes all emitted tuples among all downstream task instances. This can be useful to make optimal use of the available compute resources.[9]

In order to provide reliability guarantees, Storm offers an "acking" (acknowledgment) facility that makes sure that each tuple is successfully processed at least once. When a spout emits a tuple, it attaches an id to the outgoing tuple and will be informed by the framework, as soon as a tuple has been fully processed by all downstream bolts. In the case of errors, Storm informs the emitting spout of any tuples that failed to be processed, so it can re-emit these tuples.

In contrast to MapReduce, where processing is moved to the data, Storm has no inherent data locality as streaming data is ingested from external sources. However, processing can be arranged in a way that reduces the amount of network load incurred by the system due to data movement while still making good use of the compute resources that are available. This process is called scheduling.

### 3.2 Workload Partitioning and Scheduling in Storm

For topology edges that have a field grouping configured, Storm guarantees that all tuples that share the same value(s) for one or multiple fields, are processed by the same task instance of the bolt. To that end it hashes

[9] For more information, see https://github.com/nathanmarz/storm/wiki/Concepts

the field's values to an integer and uses the modulo function to assign a given tuple to a task instance of the receiving bolt.

As an example, consider the topology depicted in Figure 1: Whenever one of the task instances of bolt *Bo* emits a tuple, the value of the field configured in the field grouping strategy is hashed, its modulo 3 is computed, and the tuple is then sent to the task instance of *Bb* with the corresponding number (*tb1*, *tb2*, or *tb3*). We compute modulo 3 as the receiving bolt *Bb* has been configured with a degree of parallelism of 3. If there are multiple bolts in a row that all expect their input data grouped on the same field, it will be prudent to place the corresponding task instances on the same cluster machine, so that the communication overhead between the two task instances can be reduced to the passing of a pointer to an object in memory. Assuming that a storm topology keeps most of its data in memory, the topology's throughput bottleneck, therefore, is the amount of network traffic necessary to process the data. As Storm's decision of where to send a data tuple to depends on the contents of the tuples passed within the running topology, the actual communication behavior between the task instances can only be measured at runtime (or with highly specific knowledge about the data distributions that is typically not available ex ante). The communication behavior of a running topology can be thought of as a weighted directed graph, in which the weights on the nodes represent the compute resources that a task instance needs to process and the weights on the edges represent the accumulated size of all the tuples that are sent from one task to the next. We refer to this graph as the topology's *communication graph*. We describe this more formally in the next section.

### 3.3 Graph Partitioning for Scheduling in Storm

In the following paragraphs we will formally describe how we map the problem of workload scheduling in Storm to a graph partitioning problem.

**The Communication Graph** The logical view of a Storm topology can be understood as a graph $T = (B, C)$, where $B$ is a finite set of bolts and spouts connected by a finite set of connections $C \subset B \times B$. Each bolt $b_i \in B$ is configured with a degree of parallelism $dp_i \in \mathbb{N}$. Each connection $c_i \in C$ is configured with a grouping strategy $g_i \in G$.

The physical view of a Storm topology is again a graph $G = (V, E)$. Each bolt and spout $b_i \in B$ is represented by a set of task instances $V_i \in V$ where $|V_i| = dp_i$. Any two sets of vertices $V_x$ and $V_y$ are connected through at most $|V_x| \times |V_y|$ vertices. The graph is weighted. The vertex weights $vw_i \in \mathbb{R}_{>0}$ represent the amount of compute resources consumed by task instance $v_i \in V_i$. The edge weights $ew_{ij} \in \mathbb{R}_{>0}$ represent the amount of information exchanged between the two task instances $v_i$ and $v_j$.

**Graph Partitioning** A partitioning divides a set into pairwise disjoint sets. In our case we want to partition the vertices $V$ of a graph $G$ into a set of $K$ partitions. A partitioning $P = \{P_1, \ldots, P_K\}$ for $V$ separates the set of vertices such that

- it covers the whole set of vertices: $\bigcup_{k=1}^{K} P_k = V$ and
- the partitions $P_k$ are pairwise disjoint: $\bigcap_{k=1}^{K} P_k = \emptyset$

In addition, we denote (i) a *partitioning function* by $part : V \rightarrow P$ that assigns every vertex $v_i$ to a partition $P_k \in P$, (ii) a *cost function* by $cost(G, P) \in \mathbb{R}$, which denotes some kind of cost associated with the partitioning $P$ of the communication graph $G$ that is subject to optimization, and (iii) a *load imbalance factor* $loadImba(G, P) \in \mathbb{R}_{>0}$ that ensures that the workload of the tasks is evenly distributed over the machines.

We can now map our problem of minimizing the number of messages that are sent between machines to a graph partitioning problem with a specific cost function. First, we define the graph to be partitioned as the communication graph $G$ and we set $K$ to be equal to be the number of machines in our cluster. A partitioning of $G$ maps each task instance ($v_i \in V$) to exactly one machine. Second, we define a cost function for a partitioning $P$ of a communication graph $G$ as

$$cost(G, P) = \sum_{i=1}^{V} \sum_{j=1}^{V} \mathit{differentPart}(i, j) * ew_{ij}$$

where the function $\mathit{differentPart}(i, j)$ is defined as

$$\mathit{differentPart}(i, j) = \begin{cases} 1 & part(v_i) \neq part(v_j) \\ 0 & otherwise \end{cases}$$

Third, when optimizing the costs for the partitions we add the constraint that the partitions shall be balanced with respect to the computational load. More precisely, we define a load imbalance factor as

$$loadImba(G, P) = max(pw_k/apw)$$

where $pw_k$ is the summed weights of all vertices in partition $k$ and $apw$ is the average partition weight over all partitions of partitioning $P$.

In order to minimize the amount of network communication within a running Storm topology, the following optimization problem has to be solved:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & cost(G, P) \\ \text{subject to} \quad & loadImba(G, P) \leq I \end{aligned}$$

where $I$ is a the maximum imbalance we are willing to accept.

The communication graph is constructed as follows: Instead of computing the object size of each tuple, we approximate the communication load by counting the messages that that are emitted from any spout or bolt task instance. These counts are then used as a proxy for the edge weight between the sending and the receiving task instance. Assuming that each tuple that is either emitted from or received by a task instance will also incur some processing load, we sum the number of all emitted and all received tuples and use this value as the node weight in the communication graph.

All graph partitionings in this paper were computed using the METIS algorithms for graph partitioning [16] – a well established graph partitioning package.

# 4 Experimental Design

In the following paragraphs we are going to present the design of our experiments. We first list the metrics we collected to measure the performance of the systems. We then present the schedulers we evaluated and the topologies we used to measure their performance. Lastly, we present the cluster setup used.

### 4.1 Performance Metrics

A good stream scheduler maximizes the throughput of a system whilst minimizing the system load. We operationalize these measures as follows:

**Throughput** In the case of distributed streaming systems, performance is measured as throughput, which is the number of messages that the system is capable of processing. As a streaming system runs continuously, we measured throughput as the number of tuples that all spouts of a topology emit per second. We averaged this value over the whole runtime of a test run.

**Bandwidth** The most constraining bottleneck in a distributed streaming system is the the network that connects cluster nodes. As such, we measured the number of bytes that were transferred over network interfaces of the cluster machines during our experiments.

Note that we do not consider CPU load to be a constraining factor as we assume that more machines can be added to a cluster to accommodate increased load requirements. In such scenarios, internode-communication is often the principle performance bottleneck [3].

### 4.2 The Schedulers

**Default / Even Scheduler** The default scheduler shipped with Storm is called *Even Scheduler*[10]. It evenly assigns all task instances to the available worker nodes and does not take any performance metrics of the running topology into account. In this paper, we refer to this scheduler as the "Default" scheduler.

**Greedy Scheduler** Aniello et al. present two different schedulers as an alternative to the Default scheduler provided by storm in their paper in [5]. The first scheduler analyses the topologies offline and bases its scheduling decision on this offline analysis. The second scheduler they propose is a traffic-based online scheduler which takes performance metrics of the running topologies into account. The online scheduler implements a "greedy heuristic" that tries to compute an optimal schedule at

[10] https://github.com/apache/storm/blob/master/storm-core/src/clj/backtype/storm/scheduler/DefaultScheduler.clj

runtime. As the latter outperformed the former in all experiments, we only compare with the traffic-based version and refer to this scheduler as the "Greedy" scheduler. We will outline this scheduler in the paragraphs below and refer to [5] for a more detailed description.

As with our graph partitioning based scheduler, the Greedy scheduler is built around the idea of using collected performance statistics of a running topology to compute the optimal workload distribution in the cluster. Similar to our approach, information about the sending behavior of running tasks is collected. Additionally, information about how much time the threads of each task spend executing code is collected. In combination with the clock rate of the CPU, the Greedy scheduler also tries to recognize and react to system overload of compute nodes.

The algorithm works as follows: First, all task instance pairs of a running topology are sorted in descending order according to the number of messages passed between the pairs. Then, the algorithm iterates over the pairs, trying to co-locate task instances that have high communication volumes. If this co-location is not possible, because it would result in overloading one of the workers, there are nine different co-location combinations that are investigated to find the most optimal placement. An analogous strategy is then used to distribute workers across the supervisors.

We used the version of the Greedy scheduler that we downloaded from the linked sources in [5]. This version reacts to two different states that can trigger the re-scheduling process. First, scheduling can trigger when the previously computed schedule would result in an inter-machine traffic that would be lower than a certain percentage of the old schedule. This percentage value is configurable and for our experiments, has been set to $1\%$, meaning that we would always fire if there are traffic benefits to be expected. Second, scheduling of a topology can be triggered when any of the cluster machines is overloaded in terms of CPU usage. The data is collected and aggregated over windows and the scheduler does not schedule more often than every $reschedule.timeout$ seconds. For our experiments we set this value to 1 minute. The sensitivity that the scheduler shows towards load imbalances can be configured using three parameters $alfa$, $beta$, and $gamma$, for which we used the values found in [5].[11]

[11] $alfa = 0.0, beta = 0.5, gamma = 1.1$

The implementation of the Greedy scheduler we downloaded only works for spouts and bolts that have been (manually) registered. As several of Storm's internal services are based on bolts that do not contain this registration code, this implementation did not assign all task instances during the scheduling process. For this reason, we implemented a fix that evenly distributes all unassigned task instances across all workers of the cluster. These modifications, along with all the source code used for this paper, can be downloaded from https://github.com/lorenzfischer/storm-scheduler.

**Metis Scheduler** In order to test our proposed scheduling algorithm, we leveraged the graph partitioning software METIS [16]: We first use the metrics collection framework of Storm to collect the communication graph data at runtime. At a configurable timeout, the resulting graph is partitioned using METIS and the partitioning used as the workload schedule. We set this timeout value to always be the same as the corresponding timeout value of the Greedy scheduler. Note that in this paper, scheduling only happens once per evaluation run at the specified timeout. To create the schedule, we used the *gpmetis* program in its standard configuration, which creates partitions of equal size, and only changed the *-objtype* parameter to instruct METIS to optimize for total communication volume when partitioning, rather than minimizing on total edgecut. We used Metis version 5.1.0 with default partitioning (kway) and default load imbalance of 1.03. The strategy METIS uses, to compute good partitionings fast, is multilevel graph bisection: it incrementally coarsens the graph by collapsing nodes and edges to arrive at a smaller version of the graph. It then partitions this smaller graph, before it uncoarsens the graph into its original form, adapting the partitioning at each uncoarsening step to account for the newly un-collapsed vertices and edges. In this paper, we refer to this scheduler as the "Metis scheduler."

### 4.3 Topologies

We tested all three schedulers using four different topology implementations. In the following paragraphs each topology will be motivated and explained in detail.

**OpenGov Topology** The first topology represents a query over two realworld data sources which combines data on public spending in the US with stock ticker data.[12] We devised a query that would highlight (publicly traded) companies, that double their stock price within 20 days and are/were awarded a government contract in the same time-frame. The query requires the system to scan the two sources, aggregate/filter values, and finally join certain events that may have a causal relation to each using a temporal condition.

The resulting topology (Figure 2) first aggregates (A) the ticker-sourced (T) data to compute the minimum and maximum value over a time window of 20 days. It computes the ratio between these numbers (B) and then filters those solutions, where that ratio is smaller than or equal to two (F). The remaining company tickers are then joined (J[13]) with the ones that where awarded government contracts (C). The joined tuples are then sent to the output node (O). The spout nodes (T and C) read the data form HDFS and the output bolt (O) writes the results back to HDFS.

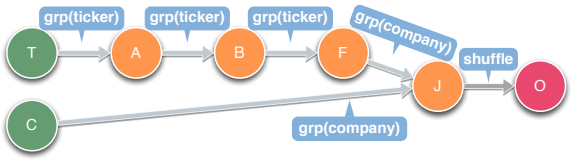



**Fig. 2.** OpenGov topology, joining ticker data (T) with public contracts (C).

When varying the cluster size, we set the parallelism for all bolts to be equal to the number of cluster machines of the test. For the spouts, we chose the combined number to be equal to the number of cluster machines and partitioned the data accordingly. The output was always sent to one single output task, so the output bolt had a degree of parallelism of one. For example, in the case of a 10 machine cluster, we configured a degree of parallelism of 10 for all bolts but the output bolt and had 5 input files for the contract and the ticker source, respectively. This setup amounts to 6 tasks for each worker machine.[14]

[12] http://www.usaspending.gov, https://wrds-web.wharton.upenn.edu/wrds
[13] We use a hash join with eviction rules for the temporal constraints.
[14] Plus one spare output bolt task.

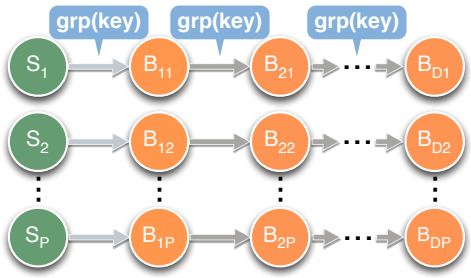



**Fig. 3.** Physical instantiation of *Parallel* and *Payload* topologies.

**Parallel Topology** The idea behind this topology was to develop a topology that would be trivial to distribute across the cluster. One way to achieve this, is to have several independent (embarassingly parallel) messaging channels: The "Parallel" topology reflects this setup. The physical instantiation of the topology is shown in Figure 3: The topology consists of one spout and a number of $D$ (depth) bolts that are all serially connected. The tuples passing between the nodes in the topology are identical throughout the topology and consist of a single 128-bit MD5 hash value which is generated in the spout. The degree of parallelism ($P$) is defined for the whole topology and defines how many instances of each spout/bolt will be instantiated in the running topology.

To keep the communication between the bolts in straight lines, each spout generates a key value that will always be routed to the same bolt instance when sent over an edge configured with the field-grouping strategy. Whenever a bolt receives a tuple, it emits a new tuple that contains the key value of the received tuple before acknowledging the received tuple. As the bolts are also connected to each other using field-grouped edges, each bolt task emits values that are routed to exactly one other bolt. This setup results in a quadratic $D \times P$ topology instantiation. Assuming that one worker node in the cluster is capable of running $D$+1 tasks, the optimal workload scheduling in this topology would be to put all $D$ bolt tasks together with their spout task on one machine and do the same for all other $P$ groups.

To prevent our bandwidth measurements to be influenced from waiting threads on overloaded worker nodes, we empirically tested how many task instances can be assigned to one worker machine without the worker being overloaded. Figure 4 shows the throughput values achieved on one

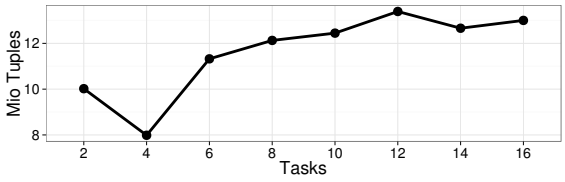



**Fig. 4.** Throughput achieved by varying the number of tasks on one cluster machine transmitting tuples of size 3KB.

worker machine running varying numbers of tasks. We note that while running fewer than 8 tasks is underutilizing the available resources, running more than 12 tasks yields no further increase in throughput. As our topologies need to run several statistics collection routines, we kept the number of tasks running on any single worker machine around 8. For this reason we chose $D = 7$, meaning that we ran 8 task instances per machine (1 spout task and 7 bolt tasks). When we varied the cluster size, we always set the value of $P$ to be the same as the number of workers.

**Payload Topology** As we anticipated the bookkeeping overhead of Storm to dominate the performance of a topology with a very small payload such as the 128-bit key of the Parallel topology, we created a variation of it: the "Payload" topology. This topology is structurally equivalent to the Parallel topology (see Figure 3). The only difference is in the tuples that are sent between tasks. In addition to the key field, which we cannot change without destroying the parallel character of the topology, we added a payload field to simulate an application that processes larger tuples, such as for example e-mail messages. We chose the size of the payload field, so that the size of the tuples amount to 3KB of data, this being the average size of an e-mail message in the publicly available Enron e-mail dataset.[15]

**Reference Topology** We have generously been provided with the code for the topologies that Aniello et al. used in [5] to evaluate their schedulers. The topology is similar in structure to the Parallel and Payload topologies and its logical representation can be found is shown in Figure 5. There is again only one spout type. However, there are three bolt

[15] http://enrondata.org/content

types, two intermediate bolts (stateful and simple) and one final "Ack-Bolt". The stateful and the simple bolt contain almost equivalent code. In the basic configuration we used, they emit integer values that increase by one for each emitted message. This results in a behavior in which a bolt, which is connected to its successor by a field-groping edge, will send an equal amount of messages to each task instance of the successor bolt in a round-robin fashion. The grouping on the edges between the spout and the bolts alternates between field-grouping and shuffle-grouping, the latter of which being a uniform distribution of the messages which results in a flooding of the network resource between two connected bolts. The size of the tuples passed between the tasks is 96bits: One integer field (32 bit) and one long field (64bit). The long field contains a timestamp which is generated in the spout an evaluated in the last (ack-) bolt of the chain. The acking facility of Storm has been turned off and replaced with a custom acking and message throttling facility, most likely to prevent the performance to be dominated by Storms bookkeeping overhead.

While the topology offers a wide array of configuration parameters, we used the topology in its default configuration and only set the minimally required parameters.[16] Similar to the Parallel topology, the Reference topology has a parameter for the number of "stages" (stage.count). It defines the length of the chain between the spout (including) and the acker bolt (excluding). We used a "stage.count" of 7, which results in a chain of 8 Storm nodes. While there are separate configuration parameters that allow a different the degree of parallelism to be configured for the spout and each bolt type, we used the same value for all of them in our evaluations (similar to the $P$ value of the Parallel/Payload topology). This strategy results again in 8 task instances per worker/supervisor when using an even schedule. Again, when varying the cluster size, we used the number of workers in the cluster as the degree of parallelism for all the topology components.

As the acking facility is normally also used to prevent buffer overflows, Aniello and his team implemented their own message throttling facility which we configured with the same value they used in [5]. While the description in [5] states that the rate at which new tuples are emitted from the spouts should be constant, we observed non-constant oscillating throughput rates. We compensate for this in our evaluations below.

[16] We refer to [5] for details.

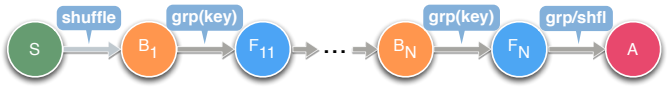



**Fig. 5.** Logical view of the *Reference* topology presented in [5].

## 4.4 Cluster Configuration

This section will give a brief overview over the cluster hard- and software we used for our experiments.

**Hardware** Many compute clusters that are in production in industry, consist of several thousand commodity computers [3]. While we did not have a cluster of this magnitude at our disposal, we made an effort to at least simulate such a cluster by connecting the work station computers, that our department offers to our students to work on, into an 80 machine Hadoop cluster. The student computers are iMac computers with Intel Core i5 CPUs (4 cores with each 2.7GHz), 8GB ram, and 250GB SSD hard drives. The iMacs are distributed over two rooms, in rows of at most 8 computers (some rows contain fewer computers). Each row is connected using a 1Gbps switches. All rows are connected over at most 2 Cisco Catalyst 4510R+E (48Gbps) switches. We scheduled our evaluations during off hours. However, we cannot exclude that there were students using the iMacs systems during the evaluations. We compensated for this by running each evaluation 3 times and taking the best value in terms of throughput as the result for the respective test run.

In version 0.23, Hadoop introduced support for other applications than MapReduce through its YARN[17] resource scheduler. For our experiments, we used the Storm-Yarn project,[18] which is an effort to run Storm inside a Hadoop cluster. In order to prevent the Hadoop cluster from going down because of a student accidentally shutting down his work station, we ran the Hadoop Job Tracker as well as the Zookeeper[19] instance on a separate machine. For this, we used a virtual machine with 4 simulated 2.6GHz CPUs.As the Greedy scheduler relies on a MySQL server to collect performance statistics, we setup a MySQL instance running on

[17] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site
[18] https://github.com/yahoo/storm-yarn
[19] http://zookeeper.apache.org

a separate virtual machine which was running on a simulated 4-core CPU with 2GHz per core.

**Software** All iMac computers ran OS X 10.9.4, having Java 1.8.0_11.jdk installed. The virtual machine running the job tracker ran on Debian 7.5 (wheezy). We used Hadoop 2.2.0 as the base system and Storm 0.9.0.1 through Storm-Yarn 1.0-alpha orchestrated by Zookeeper 3.4.5. The virtual machine running the MySQL instance was running on Debian 6.0.9 (squeeze) running MySQL 14.14.

For the Storm workers, we allocated 6GB memory on each machine, leaving 2GB for system and other processes. We configured each process running on the cluster nodes, the supervisors, to start one worker (slot) with a thread pool size of 8, two threads per core.

### 4.5 Metrics Collection

To collect the data presented in the evaluation for this paper, we used the metrics framework that storm provides. Concretely, we implemented a task hook[20] to automatically attach and start a metrics collector which collect performance information in fixed intervals. For the evaluations presented in this paper, we set this interval to 5 seconds. The metrics consumer bolt receives all metrics from all task instances and writes them to a remote Logback server.[21] These performance logs were then used for the evaluation.

The system load was measured using the Java internal MXBean facility. We used the value returned by the getSystemLoadAverage() method of the OperatingSystemMXBean class in the java.lang.management package and multiplied it by 100, to work around an incompatibility of our metrics system with floating point values. Unfortunately, the way this "system load" value is computed is operating system dependent and not clearly defined by the Java specification. It is an average value over the last minute, reflecting the cpu "load" the system has been exposed to. For this reason we compare only relative changes of the value for this performance measure and do not make any statements about the absolute values themselves.

[20] http://storm.incubator.apache.org/documentation/Hooks.html
[21] http://logback.qos.ch

The network load was measured using the unix tool “netstat”. We summed up the values for incoming and outgoing bytes for all network interfaces other than the loopback interface and used this value as the current load for the cluster node.

# 5 Experiments

We ran two sets of experiments: First, we ran all schedulers with all four topologies, across a series of different cluster configurations. The results of these experiments are presented in section 5.1. To investigate two observations we made in this first set of experiments in more detail, we ran a second set of experiments on which we report in 5.2.

## 5.1 Bandwidth and Throughput Experiments

We ran all four topologies using all three schedulers three times each using different cluster size configurations varying between 10 and 80 machines. Each configuration was running for 300 seconds. For the Metis and Greedy schedulers, the re-scheduling timeout was set to 60 seconds, after which the computed schedule is applied to the running system, i.e. task instances are re-assigning to potentially different machines. Note that state handling in Storm is the responsibility of the topology developer. As the time it takes for all the machines to start all required processes can vary, we removed the first and the last 60 seconds from the data, leaving us with data for 180 seconds of log data for each run. This also removes the metrics data collected before the re-scheduling occurs.

In figure 6 we present an overview over the achieved reduction of bandwidth consumption as well as the increased throughput for both schedulers compared to the default scheduler as a baseline. As the performance in a cluster system with up to 80 nodes can vary in each run, we computed the relative changes using the best-of-three result in terms of throughput for each configuration. We show max-min-avg statistics as well as a discussion of these results in the following paragraphs.

**Reduced Network Load through Scheduling** Studying figure 6, we observe that using our Metis scheduler results in substantial reductions in network usage in almost all configurations. The greatest improvements were achieved with the Payload topology, where we measured reduced

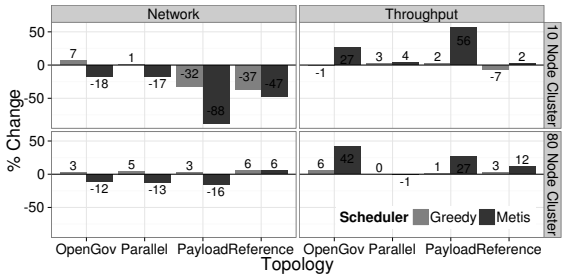



**Fig. 6.** Baseline comparison in regards to bandwidth consumption and throughput for the Greedy and Metis scheduler against the Default scheduler on a 10 node cluster as well as an 80 node cluster.

network load of up to 88% for the 10 node cluster configuration. The greatest average improvements in terms of network load were measured for the Payload topology (33.75%), the lowest using the Parallel topology (14%). All of these improvements are statistically significant with $p < 0.05$. As we have non-normally distributed data, we employed a Mann-Whitney test to calculate significance.

Applying the Greedy scheduler yields bandwidth savings as well. However, we observed many instances in which the gains are either not very large, or not statistically significant. We investigated this issue and noticed that for problems of size 20 and more, the Greedy scheduler often did not reschedule a topology. The Greedy scheduler exhibited greatest decreases in network usage for the Reference topology (9.75% on average). From these observations it seems that the Greedy scheduler is better suited for topologies with varying workloads and that our topologies, which show mostly static throughput rates, were not well suited to take advantage of this scheduler.

**Increased Throughput through Scheduling** Even though we cannot make a statement about the true bottleneck of the system being CPU, network, or system latency, we observe higher throughput when using our Metis scheduler compared to the Default scheduler in all cases, and in most cases compared to the Greedy scheduler. Comparing the best-of-three evaluation runs, we observed a significant ($p < 0.05$) improvement over the Default scheduler in terms of throughput in all configurations and over all topologies. Compared with the Greedy scheduler, we observed significant ($p < 0.05$) throughput increases in all but one case (we

measured a 1% lower throughput using Metis compared to the Greedy scheduler in the 80-node Parallel topology run). Other instances where Metis performed worse than Greedy were not statistically significant and they were measured for the Reference topology using a 50, 60, and 70 node cluster. Table 1 serves as a summary: The greatest average improvement (across all cluster configurations) were measured with the Payload and the OpenGov topologies (+52% and 19% on average). We noticed that tuple improvements tend to be higher in topologies in which the tuple sizes are large. Tuple sizes of the four topologies are listed in Table 2. We investigate this in more detail in Section 5.2. We also measured the variation between the three evaluation runs for each configuration: The greatest min-max-spread can be observed for the payload topology. This intuitively makes sense, as when optimizing the schedule of topologies with large tuple sizes, one would expect to achieve greater improvement for good schedules, but also greater variability between evaluation runs. As Figure 7 shows, however, our Metis scheduler outperforms both other schedulers even if we were to take the worst-of-three performance in most cases.

| Scheduler / | Default | | Greedy | | | Metis | | | |
|---|---|---|---|---|---|---|---|---|---|
| Topology | a | m | a | m | id | a | m | id | ig |
| Parallel | 5 | 10 | 21 | 53 | 0 | 13 | 29 | 6 | 6 |
| Payload | 20 | 38 | 23 | 53 | 12 | 26 | 70 | 52 | 38 |
| OpenGov | 4 | 11 | 5 | 12 | 0 | 9 | 17 | 19 | 19 |
| Reference | 9 | 16 | 9 | 22 | 1 | 5 | 12 | 3 | 2 |

**Table 1.** Average (a) and maximum (m) min-max-spread over three runs, in terms of throughput, as well as the improvement compared to the Default Scheduler (id) and the Greedy scheduler (ig), computed over cluster configurations ranging from 10 to 80 nodes (percentage values).

| Topology | Size | Contents |
|---|---|---|
| Reference | 12B | ID + System Time |
| Parallel | 16B | single 128bit hash |
| Opengov | 50B - 110B | Dates, Prices, and Names |
| Payload | 3072B | Average Email Size |

**Table 2.** Typical Tuple Sizes (in bytes)

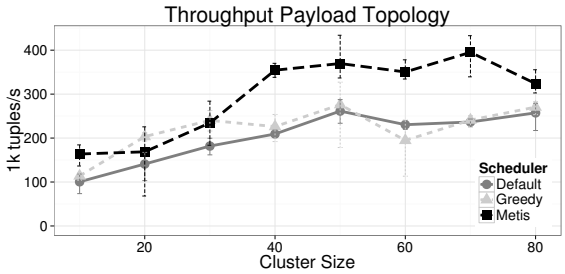



**Fig. 7.** Payload Topology: min-max-avg throughput.

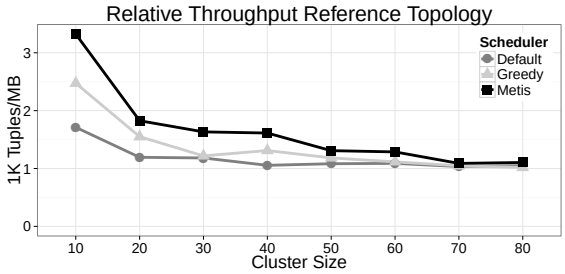



**Fig. 8.** Relative throughput of the Reference topology.

The throughput of the Reference topology is controlled by a throttling mechanism that is dependent on configuration parameters and not on how good use the system makes of the resources. While Aniello et al. write in [5], that the output rate of the spouts should be constant, we did not observe this in our evaluations. For this reason, we plotted the bandwidth in relation to the tuples that were emitted from the spouts in Figure 8. We can see that both, our METIS-based scheduler as well as the Greedy scheduler achieve better performance than the Default scheduler. We observe that the Metis scheduler dominates this comparison as well, showing performance superiority over the Greedy and Default schedulers of between 34% respectively 95% (for cluster size 10) and 8% respectively 5% (for cluster size 80).

**Balanced Workload** One implicit assumption we make in this paper is, that by assigning workload based on simple tuple counts, we can achieve a balanced workload distribution in terms of CPUload across a cluster. To investigate this, we plotted the average CPU load per worker in Figure

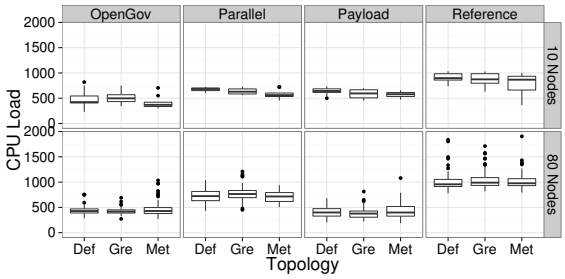



**Fig. 9.** Mean workload distribution in terms of CPU load per worker for the best of three runs using the Default (Def), Greedy (Gre) and Metis (Met) scheduler for all topologies on clusters with 10 and 80 nodes.

9.We observe that using our Metis scheduler did result in distributions that are similar to the distributions achieved using the Default and the Greedy scheduler. Studying the lower row of Figure 9 we note that the number of outliers (machines with radically higher CPU load) are in general higher. We assume that this is due to the fact, that with 80 machines in a cluster, it is very likely that some other process such as an automatic update or backup happened during the time we ran our evaluations.

## 5.2 Tuple Size & Fault Tolerance Experiments

While the savings in bandwidth achieved through scheduling are substantial, we noticed that they do not necessarily correlate with equally substantial increases in tuple throughput. In particular, we observe that the throughput increases through scheduling are larger with increasing tuple-size. We suspected that this is due to bookkeeping overhead induced by Storm's acking facility which provides fault tolerance. To investigate this we conducted a second set of experiments: In the next paragraph, we vary the tuple size systematically and observe changes in the throughput and in section 5.2 we examine the impact of the acking facility on the bandwidth consumed by the topology.

**Tuple Size vs. Throughput** From the three topologies whose throughput is not externally controlled (Parallel, Payload, OpenGov), we see that the improvements in bandwidth usage and throughput tend to be higher, the bigger the size of the tuples that are processed are. In Figure 10, we plotted the average throughput over all spouts in a 40-machine cluster and

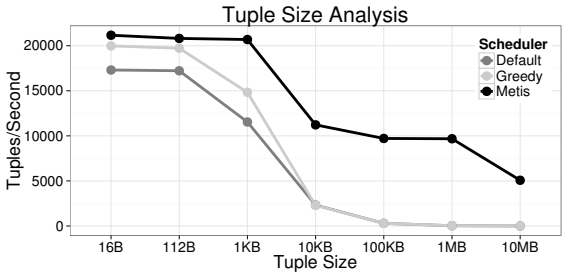



**Fig. 10.** Throughput measured in tuples per second per spout with varying tuple sizes on a cluster of 40 machines on the Payload topology.

varied the payload of the tuples processed. To prevent out-of-memory exceptions, we used a shortened topology with only depth 3 for these experiments. The experiment confirms our observation that the bigger the payload, the more we can improve throughput by adequate workload scheduling. Our METIS-based scheduler outperforms both, the Greedy and the Default scheduler, by a large margin. For tuple sizes of 10KB and up by a factor more than four.

**Fault Tolerance vs. Throughput** The throughput of a topology is dependent on how fast messages can be processed. One component that makes up for the processing time is, how fast messages can be fully processed (and acknowledged). To assess the degree to which the facility providing fault tolerance in Storm (the acking facility) had an impact on the performance, we ran a set of experiments with the acking facility turned on and off. In Storm, throughput is throttled using the acking facility: The user defines an upper bound for the number of unacknowledged tuples per spout.[22] As this mechanism is unavailable when the acking facility is turned off, we chose a constant rate of 4000 tuples and resolved to measuring the amount of bandwidth incurred on the network during the test runs. The results are shown in Figure 11, where we plot the bandwidth usage of the Payload topology using different cluster sizes. For these evaluations we chose a medium size payload of 112 bytes (corresponding to the typical OpenGov message size) and compared the performance of the Default scheduler with the performance of our graph-partitioning-based approach. As we can see in the figure, the

[22] See the Storm option $max.spout.pending$

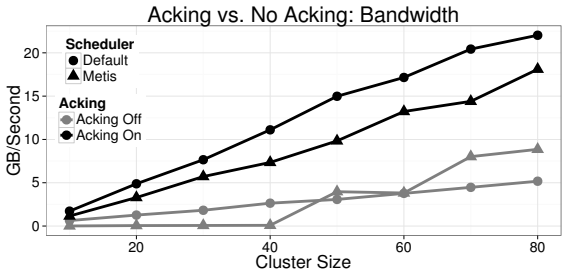



**Fig. 11.** Bandwidth usage with a topology emitting a constant rate of 4000 (112B) tuples/s, with the acking facility turned on and off.

acking facility of Storm uses a substantial amount of bandwidth that we cannot compensate for by mere scheduling.

In order to understand these results, we need to elaborate on how Storm's acking facility is implemented. Acking in Storm works as follows: The bookkeeping of the acking facility is done by a system internal Acker bolt. Whenever a spout emits a message, a random identifier is generated and attached to the tuple.When a task instance receives and processes a tuple, it acknowledges the receipt of the tuple with the responsible acker task instance. The receiving task instance can generate one or multiple new tuples in response to processing a received tuple. When emitting a tuple, it "anchors" the new tuple by attaching the id of the "parent" tuple to the emitted tuple, building a tuple tree in the process. The component responsible of keeping track of these tuple trees is the acker bolt. By default, the acker bolt has a degree of parallelism equal to the number of workers in the cluster, so there is one acker for each worker. When a spout emits a tuple, it sends the message id of that tuple to the responsible acker task instance. This acker instance will keep track of all emitted tuples and inform the emitting spout task instance, when the tuple tree for the emitted tuple has been fully acknowledged. The decision over which acker instance is responsible for which tuple is made in the same fashion the regular scheduler is implemented, by first hashing the message id and by taking the remainder of a division by the number of acker instances (worker nodes). As the spout task instances generate these message ids in a random fashion, each acker will be responsible to track the tuple tree of various different spout task instances. While this leads to an equal distribution in terms of message ids to ackers,

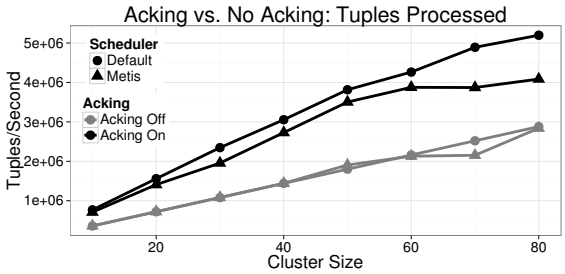



**Fig. 12.** Number of tuples processed within the whole topology at constant throughput rate of 4000 tuples/s, with the acking facility turned on and off.

it also requires that the acking facility needs to send network messages across the whole cluster in order to function, regardless of what scheduler is used. We show this effect in Figure 12, in which we plotted the total number of tuples processed by the whole Payload topology that, again, emits in each spout at a constant rate of 4000 tuples per second: The acking facility effectively doubles the number of messages that need to be processed by the system.
As a consequence, the random choice of the message ids prevents optimization of the communication workload produced by the acking facility. This in turn explains, why topologies with small tuple sizes do not profit from scheduling to the same extend as topologies with large tuple sizes: When tuples are small the non-optimizable acking overhead uses a large fraction of the network bandwidth. As the strategy of uniformly choosing message identifiers in Storm guarantees an equal workload distribution of the bookeeping work, one solution to this problem could be to not acknowledge every single tuple separately, but to batch multiple tuples into mini-batches, an approach recent additions to the Storm framework have taken.

reduce

## 6 Conclusions & Limitations

In this paper we presented a graph partitioning based scheduling approach for task-parallel distributed stream processing systems. We implemented our approach as a scheduler for the Storm realtime computation framework using the METIS graph partitioning software and eval-

uated its performance against two state of the art schedulers. We have shown that a workload scheduler based on graph-partitioning can substantially and significantly lower network utilization while at the same time increase overall throughput. Our scheduler showed superior performance in almost all experiments with decreases of network bandwidth of up to 88% and increased throughput values of up to 56%, respectively. We have shown that this approach scales well to setups with up to 80 machines and 360 task instances. As our approach builds on a proven graph partitioner that scales to graphs with millions of edges [16], we dare to hope that our scheduler can scale to much larger setups. The fact that the improvements increase with tuple size along with the observation that fault tolerance through acking is expensive suggests that the recent trend towards mini-batching in task-parallel distributed systems is likely to profit even more from throughput-based online scheduling.

Our investigation is hampered by the following limitations. First, we only approximate worker machine load by counting the number of tuples that are received and emitted by the tasks running on the machine. We believe that more accurate performance metrics, similar to the ones Aniello et al. use in [5], could provide more precise statistics that may result in better schedules. Second, our current approach does not take into account some computational resources such as available memory or disk space. Such an extension seems desirable and straightforward via a more complex load function of our communication graph. Third, our findings' external validity is somewhat hampered by the topologies chosen. In the real world, topologies are oftentimes somewhat more complicated and experience bursty load (see [10]).

Even in the light of these limitations, this paper has presented and evaluated a novel approach for workload scheduling in a task-parallel distributed streaming system. From our findings, we believe that it presents a step towards a holistic solution for scheduling that leverages a proven optimization approach for realistic cluster sizes.

# References


[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proceedings of the 2005 CIDR Conference*. Citeseer, Citeseer, 2005. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.7039&rep=rep1&type=pdf.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Whittle Sam. Millwheel : Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6 (11), 2013. URL http://research.google.com/pubs/archive/41378.pdf.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*. Association for Computing Machinery (ACM), 2008. doi: 10.1145/1402958.1402967. URL http://dx.doi.org/10.1145/1402958.1402967.

[4] Alex Aletà, Joseph M. Codina, Jesús Sanchez, and Antonio González. Graph-partitioning based instruction scheduling for clustered processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001. URL http://dl.acm.org/citation.cfm?id=563998.564019.

[5] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems - DEBS '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2488222.2488267. URL http://dx.doi.org/10.1145/2488222.2488267.

[6] Bradford L. Chamberlain. Graph partitioning algorithms for distributing workloads of parallel computations. Technical report, 1998.

[7] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism. *Proc. VLDB Endow.*, 3(1-2):48–57, sep 2010. doi: 10.14778/1920841.1920853. URL http://dx.doi.org/10.14778/1920841.1920853.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - OSDI '04*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[9] Lorenz Fischer, Thomas Scharrenbach, and Abraham Bernstein. Scalable linked data stream processing via network-aware workload scheduling. In *International Workshop on Scalable Semantic Web Knowledge Base Systems - SSWS '13*, 2013. URL http://ceur-ws.org/Vol-1046/.

[10] Shen Gao, Thomas Scharrenbach, and Abraham Bernstein. The CLOCK data-aware eviction approach: Towards processing linked data streams with limited resources. In *Proceedings of the 11th Extended Semantic Web Conference - ESWC '14*, pages 6–20. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-319-07443-6_2. URL http://dx.doi.org/10.1007/978-3-319-07443-6_2.

[11] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, dec 2012. doi: 10.1109/tpds.2012.24. URL http://dx.doi.org/10.1109/TPDS.2012.24.

[12] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*. Association for Computing Machinery (ACM), 2014. doi: 10.1145/2611286.2611294. URL http://dx.doi.org/10.1145/2611286.2611294.

[13] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/Eu-*

*roSys European Conference on Computer Systems 2007 - EuroSys '07*. Association for Computing Machinery (ACM), 2007. doi: 10.1145/1272996.1273005. URL http://dx.doi.org/10.1145/1272996.1273005.

[14] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*. Association for Computing Machinery (ACM), 2009. doi: 10.1145/1629575.1629601. URL http://dx.doi.org/10.1145/1629575.1629601.

[15] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. Sqpr: Stream query planning with reuse. *Proceedings - International Conference on Data Engineering*. ISSN 10844627. doi: 10.1109/ICDE.2011.5767851.

[16] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, jan 1998. doi: 10.1137/s1064827595287997. URL http://dx.doi.org/10.1137/S1064827595287997.

[17] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009. URL http://dl.acm.org/citation.cfm?id=1657002.

[18] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2517349.2522738. URL http://dx.doi.org/10.1145/2517349.2522738.

[19] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2517349.2522716. URL http://dx.doi.org/10.1145/2517349.2522716.

[20] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering - ICDE '06*. Institute of Electrical & Electronics Engineers (IEEE), 2006. doi: 10.1109/icde.2006.105. URL http://dx.doi.org/10.1109/ICDE.2006.105.

[21] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2465351.2465353. URL http://dx.doi.org/10.1145/2465351.2465353.

[22] B. Thirumala Rao and L.S.S. Reddy. Survey on improved scheduling in hadoop mapreduce in cloud environments. *International Journal of Computer Applications*, 34(9), 2011.

[23] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2370816.2370826. URL http://dx.doi.org/10.1145/2370816.2370826.

[24] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware '08*, 2008. URL http://dl.acm.org/citation.cfm?id=1496950.1496970.

[25] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *21st International Conference on Data Engineering - ICDE '05*. Institute of Electrical & Electronics Engineers (IEEE), 2005. doi: 10.1109/icde.2005.53. URL http://dx.doi.org/10.1109/ICDE.2005.53.

[26] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 775–786. VLDB En-

dowment, 2006. URL http://dl.acm.org/citation.cfm?id=1182635.1164194.

[27] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing - HotCloud '10*, pages 10–10. USENIX Association, 2010. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.

# Machines Tuning Machines:
## Configuring Distributed Stream Processors with Bayesian Optimization

This chapter is based on a paper that has been accepted at and published in the proceedings of the *2015 IEEE International Conference on Cluster Computing (IEEE Cluster 2015)*.

# Machines Tuning Machines:
## Configuring Distributed Stream Processors with Bayesian Optimization

Lorenz Fischer, Shen Gao, and Abraham Bernstein

[1] DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
lfischer@ifi.uzh.ch
[2] shengao@ifi.uzh.ch
[3] bernstein@ifi.uzh.ch

**Abstract.** Modern distributed computing frameworks such as Apache Hadoop, Spark, or Storm distribute the workload of applications across a large number of machines. Whilst they abstract the details of distribution they do require the programmer to set a number of configuration parameters before deployment. These parameter settings (usually) have a substantial impact on execution efficiency. Finding the right values for these parameters is considered a difficult task and requires domain, application, and framework expertise.
In this paper, we propose a machine learning approach to the problem of configuring a distributed computing framework. Specifically, we propose using Bayesian Optimization to find good parameter settings. In an extensive empirical evaluation, we show that Bayesian Optimization can effectively find good parameter settings for four different stream processing topologies implemented in Apache Storm resulting in significant gains over a parallel linear approach.

## 1 Introduction

The configuration of a distributed system is crucial for both good performance and to prevent system failures [31]. Many modern distributed programming frameworks offer a wide range of configuration parameters for tuning purposes. The performance of a system deployment depends on the interplay between all parameters with the implementation of the application logic, the underlying hardware, as well as the data that is processed by the system. Hence, choosing suitable configuration parameters given a system implementation and associated infrastructure can be difficult and requires expert knowledge of both the problem domain and the technology used to build the system. Even experts require careful experimentation as the interactions between different parameters are hard to predict.[4]

[4] www.slideshare.net/miguno/apache-storm-09-basic-training-verisign

To address this tedious manual parameter experimentation, this paper proposes an automated process based on Bayesian Optimization for finding optimal parameter configurations. Specifically, we present empirical results from a series of experiments in which we evaluated the suitability of Bayesian Optimization for the configuration of distributed stream processing systems built using Apache Storm.[5] Our contributions are:

- We present an *auto-configuration approach for distributed stream processing systems (SPS) using Bayesian Optimization*.
- We provide an *extensive empirical evaluation* showing the effectiveness of our approach on a cluster of 80 machines (320 cores) running Storm topologies (applications) of varying sizes and characteristics.
- We introduce a reusable *benchmark consisting of a set of operator graphs as well as generation approach*.

The remainder of this paper is organized as follows: next, we first present related work. We then describe the system used for the evaluations and the give an introduction to Bayesian Optimization in Section 3. Our experimental setup and results are presented in Sections 4 and 4.2, respectively. We close with conclusions in Section 5.

## 2 Related Work

### 2.1 Configuration of distributed stream processing systems

The problem of how to configure distributed (stream) query systems [20] and how to react to dynamically changing properties of stream processors [10] has been extensively studied in the past decade. Often, cost models have been proposed to capture complexities of these systems [7, 9] to optimize the use of resources [7, 18] or query execution [16, 24]. Others have applied Covariance Matrix Adaption (CMA) [23] or searched the parameter space using an experimentally constructed parameter dependency graph [31]. The problem we tackle in this paper differs from the problem of cost-model based solutions in two aspects: first, we do not aim at changing the structure of the streaming application (or the query), but focus on tuning of configuration parameters to make the execution more efficient. Second, we do not attempt to build a complete (closed-form) mathematical model of the system, but treat the application as a

[5] https://storm.apache.org

blackbox function that we optimize using empirical sampling. In contrast to the approaches presented in [23] and [31], we employ a probabilistic bayesian approach.

Similar to our goal, some studies have focused on one specific parameter: the degree of parallelization. One line of work investigates auto-parallelization – the process of automatically choosing the degree of parallelism for operators in a task graph [26]. It has been extensively studied in the realm of IBM's System-S [30] as a theoretical model. Schneider et al. [25, 15] extended these results and presented an algorithm to dynamically change the workload on operators in response to changes in the incoming data stream. In contrast to pure auto-parallelization, our approach treats the parallelism of each operator of the topology as only one of many system parameters that need to be tuned. Note that we do not cover dynamic auto-parallelization as we assume mostly static workloads.

### 2.2 Applications of Bayesian Optimization

Bayesian Optimization [21] is a probabilistic technique to optimize systems with unknown cost functions. It has successfully been applied in cases where the performance of systems is strongly dependent on configuration parameters, and no mathematical closed-form cost model is known such as finding good hyperparameter settings in machine learning problems (e.g., classification [5, 28, 27] or feature selection [28]). There are several Bayesian Optimization frameworks (e.g, Spearmint[6] [27], SMAC[7][19], HyperOpt[8], or BayesOpt[9]) available for research. We are not aware of any previous work that has investigated the applicability of Bayesian Optimization for the configuration of distributed systems or for distributed stream processing systems in particular.

## 3 System Description

This section describes how we employ Bayesian Optimization to configure a distributed stream processing system based on the Storm distributed

[6] https://github.com/HIPS/Spearmint
[7] http://www.cs.ubc.ca/labs/beta/Projects/SMAC
[8] http://jaberg.github.io/hyperopt
[9] http://rmcantin.bitbucket.org/html

realtime computation framework. We first give a short introduction into Storm and Trident; the two technologies we use to implement our experiments. We then formally describe the process of Bayesian Optimization before presenting Spearmint, the optimizer used in the experiments.

## 3.1 Distributed Stream Processing with Storm

Many distributed computation frameworks have been proposed in recent years. One representative of such a framework aimed at distributed stream processing is the Storm framework. In contrast to batch-based distributed systems such as Apache MapReduce,[10] Storm ingests data continuously. As in MapReduce, a Storm application allows the user to partition the data and to distribute parts of the processing across a compute cluster.

A Storm application—a *topology*—is a directed graph consisting of spout and bolt nodes as depicted in Figure 1 on the left. Spouts emit data to downstream nodes. Bolts consume data from upstream nodes and emit data to downstream nodes. Spout nodes are typically used to connect a Storm topology to external data sources such as queues, web-services, or file systems. For each spout and bolt, the programmer defines how many instances of this node should be created in the physical instantiation of the topology – the task instances. This results in a physical topology depicted on the right of Figure 1, which is different from the logical representation. The parameter used to define the degree of parallelism of a node is called a *parallelism hint*, as Storm may change these hints for consistency purposes. The task instances, or *tasks*, are distributed across all machines of the compute cluster to which a topology has been assigned. Each edge in the topology graph defines a *grouping strategy* according to which messages that pass between the nodes—the *tuples*—are sent to downstream nodes.

Tuples are lists of key-value pairs. The programmer defines the tuple format for each edge of the topology (e.g. field1=query_terms, field2=browser_cookie, field3=timestamp). This format cannot be changed at runtime. Different grouping strategies provide different guarantees. For example, the field grouping strategy guarantees, that all tuples that share the same value in one or multiple configurable fields are sent to the same task instance.

[10] http://hadoop.apache.org

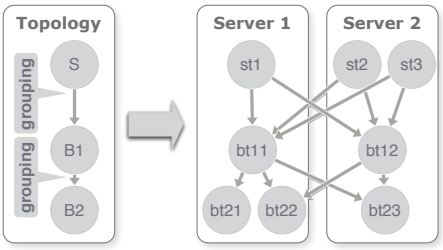



**Fig. 1.** Logical (left) and physical (right) representation of a topology. The spout (S) and bolt nodes (B1, B2) are instantiated as spout task instances (st1-st3) and bolt task instances (bt11-bt23) across two servers.

Higher level operators such as aggregators, state handling, functions, and filters are provided by *Trident*, a programming framework that is part of the Storm distribution. Further, Trident may combine multiple operators into larger units. In such cases, Storm overrides the parallelism-hints specified by the programmer in order to prevent frequent reshuffling of data across the network. This is similar to the *SPADE* system [14], which also fuses several operators into one processing element (PE) in System-S. In Trident, tuples are processed in mini-batches, offering consistency guarantees on a per-batch basis.

Having introduced the basic building blocks of a Storm/Trident application, we will describe the various ways in which such an application can be configured and tuned in the next section.

### 3.2 Configuration Parameters

Storm offers a number of configuration parameters that allow the programmer, as well as the system administrators, to configure various aspects of the system. Table 1 lists the parameters that we used in our evaluations: parameters that are most commonly tuned are the already mentioned parallelism hints, the batch size, and the batch parallelism[11] of a topology. Trident guarantees consistency on a per-batch level. This means that multiple batches can run at the same time, which can increase overall performance. Note that the "parallelism hints" parameter is not one single value, but a list of values that contains one number for each node of the topology. Hence, for topologies of greater size, the parameter space, naturally, becomes large. Other parameters that we included

[11] Batch parallelism is also called pipeline parallelism in the literature.

in this study are concurrency related parameters such as the size of the thread pool available to each worker, the number number of threads each worker starts to receive messages, and the degree of parallelism of the "acker" system bolt, i.e. the number of "acker" task instances, that Storm uses for its bookkeeping facility.

| Parameter | Description |
|---|---|
| Worker Threads | Number of threads per worker |
| Receiver Threads | Number of receiver threads per worker |
| Ackers | Number of acker tasks |
| Batch Parallelism | Number of batches being processed in parallel |
| Batch Size | Number of tuples in each batch |
| Parallelim Hints | Number of task instance to create for operators |

**Table 1.** Configuration parameters.

While any single one of these configuration options impacts the runtime behavior, overall performance is a result of the combination of all of these parameters working together. For example, consider the situation in which we set the parallelism hint of the spout in the topology depicted in Figure 1 to 10, but the parallelism hint for all bolts to 1. In this situation, the performance will most likely be bottlenecked by the code in the bolts of the topology. If, on the other hand, the parallelism hints for the bolts are set to 100, the new bottleneck would most likely be the code in the spout node. Similarly, there are interactions between the parameters for the batch size and the batch parallelism. Because these interactions are not only dependent on the values of these parameters themselves, but also on other aspects such as the available network infrastructure, disk speed, or availability of memory storage, making predictions about the resulting performance of the overall system is difficult. Additionally, framework properties, such as the automatic operator fusion of Trident, further obfuscate the impact of any single parameter. To tackle the problem of choosing good configuration parameters, we investigate the possibility of having a computer program choose these parameters. To this end, we employ the technique of Bayesian Optimization.

### 3.3 Bayesian Optimization

In this sub-section, we give a short introduction to Bayesian Optimization. We refer to [6] and [27] for a more detailed introduction into the

topic. Bayesian Optimization has first been proposed by Jonas Mockus as an optimization strategy for situations in which the objective function is a non-convex blackbox function [21] (i.e., a function for which no closed-form solution or derivative is known). The function is assumed to be Lipschitz-continuous (i.e., smooth and does not change dramatically). Also, sampling the function is assumed to be costly, either in terms of time or money. Thus, it can pay off to invest computational resources into computing the point in the parameter space where to sample next. For our domain, we assume the function to be the actual system performance of our distributed stream processor, given all the configuration parameters chosen. Obviously, given the black-box nature of the system, no mathematical representation exists, and determining the value of the function given certain parameter settings is achieved by running the system on a cluster with these settings and, hence, is costly. The process of choosing the next set of parameters is conducted using a Bayesian approach, which combines our prior assumptions about the function with the observed performance from previous runs. Borrowing the notation from [6] we can describe this formally as follows:

$$P(M|E) \propto P(E|M)P(M)$$

The probability distribution over our model M (our blackbox function) given some observed evidence E (our sampling runs) is proportional to the likelihood of E given the model times the prior probability of the model. Thus, we reason about the likelihood of observing the results of an evaluation run, given our prior beliefs about how the system would change in response to parameter modifications. Using the results of each evaluation run, a posterior distribution $P(M|E)$ is computed and integrated into the model. The decision of where to sample next is made by maximizing an acquisition function. There are various ways in which this acquisition function can be modeled. Often, Gaussian Processes are used to model the noise within the acquisition function. The purpose of the acquisition function is to balance the tradeoff between exploration and exploitation. The goal is to sample the next measurement in a region where either the uncertainty of the expected performance is high, the expected performance is high, or both.

More formally, again borrowing the notation from [6], we can describe the process as follows: Bayesian Optimization is an iterative process in which we sample an objective function repeatedly. We define $x_t$

to be the $t$-th sample and $y_t = f(x_t) + \epsilon_t$ to be the measured performance of our algorithm for run $t$, where $f$ is our blackbox target function and $\epsilon_t$ is noise, which is typically assumed to be Gaussian. Our prior believes about $f$ can be expressed as a prior distribution $P(f)$. We then collect observations (measured samples) and add them to the set $D_{1:t} = \{x_{1:t}, y_{1:t}\}$ of all evidence to date. In each step, we update our posterior belief with the newly collected evidence:

$$P(f|D_{1:t}) \propto P(D_{1:t}|f)P(f)$$

The new evidence is used to fit a Gaussian Process (GP) that describes our prior believes of how $f$ is distributed:

$$f(x) \sim GP(m(x), k(x, x'))$$

where $m$ is the mean function at position $x$ and $k$ is the covariance function depending on $x$ as well as on the closest perviously sampled point at $x'$. The result is a function estimating the expected performance of any parameter value combination given some confidence interval. An acquisition function $u(x|D)$ is built using these two parameters (expected performance and confidence intervals) that are derived from the data $D$. The goal of the acquisition function is to create a tradeoff between exploration (try points with high uncertainty/variance) and exploitation (try points with a high expected performance). Hence, the next sample point $x$ is determined by maximizing $u(x)$ (i.e. the $x$ where the tradeoff between exploration and exploitation is optimal):

$$x_{t+1} = argmax_x u(x|D_{1:t})$$

There are several different ways of defining the acquisition function such as *Probability of Improvement* (PI), *Expected Improvement* (EI), or *GP Upper Confidence Bound* to name the most common ones. In this paper, we use Expected Improvement [22], as it provides a good tradeoff between exploration and exploitation and it is the method implemented in *Spearmint*, the toolkit we use in our experiments. The Expected Improvement acquisition function proposed by Mockus [22] is defined as:

$$x_{t+1} = argmax_x \mathbb{E}(max\{0, f_{t+1}(x) - f^{max}\}|D_{1:t})$$

where $f^{max}$ is the best solution in the first $t$ samples, so the next $x$ would be chosen at the position, where the expected improvement between the

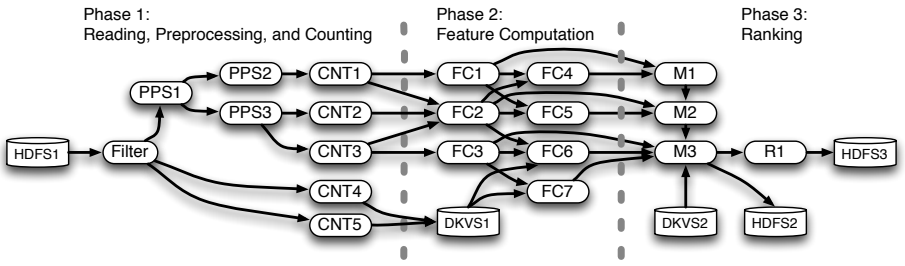



**Fig. 2.** High-level architecture of Sundog (from [13]).

new sample point ($f_{t+1}(x)$) and the current best sample point ($f^{max}$) is maximized. In contrast to the original optimization problem (our black-box function), we can derive a closed-form expression for this problem, which can then be maximized using an analytical approach. We refer to [6] for all details.

As already mentioned in section 2.2, there exist a number of freely available programming toolkits that implement Bayesian Optimization. In this project, we leverage Spearmint for the following reasons: first, it showed good performance in comparison with other main-stream Bayesian Optimization frameworks [11]. Second, it is well documented and its source code is openly available. Last, it supports pausing and resuming the optimization process, a feature that turned out to be important in our evaluation setup.

# 4 Experimental Design

To evaluate the usefulness of Bayesian Optimization for parameter configuration of an SPS, we conducted a series of experiments using one real world application and three synthetic topologies. This section describes these topologies and the experimental setup.

## 4.1 Sundog: A Real World Topology

The first topology is a modified version of the Sundog entity ranking system [13]. Entity ranking systems consume search logs, tweets, etc., and rank search results based on co-occurence statistics. Figure 2 gives a a high level overview of the topology: in the first phase, input data is read from the *Hadoop Distributed Filesystem* (HDFS). Then, all input lines that do not contain at least one term of a predefined dictionary are

filtered out. From this reduced data stream, statistics such as the number of term occurrences are built. These values are stored in an external distributed key-value store (DKVS1) to enable access from all compute tasks in later phases of the processing pipeline. For other statistics, we first build entity pairs from the terms in a series of preprocessing steps (PPS1-3) to count the number of search events and unique users for each entity and entity pair. Wherever possible, the relevant data is partitioned to allow parallelization to multiple compute nodes. The second phase consists of computing the actual feature metrics from the counter values (FC1-7). In the final phase (phase 3), the computed features are merged and complemented with semi-static features that are read from a table in the distributed key-value store (DKVS2 in Figure 2). Semi-static features such as the semantic type of an entity do not change often (or not at all). After merging all features, a score is computed for each entity pair using a decision tree.

While the original system is processing search log data, the modified version we used for the experiments presented in this paper uses a dump of the common crawl data[12] as input and we replaced calls to the distributed key-value store with dummy methods which always return 1. Even though these changes invalidate the actual rankings that the system computes, they do not change the workload characteristics of the original system.

### 4.2 Synthetic Topologies

To gain insight into how well our proposed optimization strategy generalizes to other topology designs, we generated a series of synthetic Storm topologies and evaluated the performance gains with each of them. To this end, we used the widely used graph generator *GGen* [8] to generate three topologies. We then modified these graphs by assigning different values for time and resource complexities to each vertex of the graph.

Processing pipelines in Storm typically consist of several tasks, some of which can run independently in parallel, while others need to wait for input data from upstream nodes. For this reason we generated "layer-by-layer" graphs, as motivated in [29]. In layer-by-layer graphs, nodes are grouped in layers. Nodes in the same layer only have links to nodes of downstream layers, but no links to nodes of the same layer. As we want

[12] http://commoncrawl.org

to test each graph over the course of 60 or more sampling runs, each run taking two to ten minutes, while varying node attributes of the graph such as necessary processing time or the use of constrained resources, we could only afford a small number of base graphs/topologies. To ascertain typical topology sizes we reviewed the literature (see Table 3): we found that most currently published topologies have fewer than 60 vertices, whilst enterprise-grade application may have up to 100 components [17]. Hence, we generated topologies of three different sizes having 10, 50, and 100 vertices.

To get valid SPS and comparable graphs, we ensured that (1) all vertices of the graph are connected to at least one other vertex in the graph and that (2) the average out-degree across the whole graph is approximately constant in all the produced graphs. Since GGen allows choosing (i) the number of vertices in the graph, (ii) the number of layers in the graph, and (iii) the probability of vertex to connect to vertices of different downstream layers only, we picked parameters that would fulfill these constraints as listed in Table 2. The table reports on the configuration parameters the number of vertices, layers, and probabilities to connect to vertices of the next layer as well as the typical graph statistics such as the number of edges, spout vertices (or sources), the number of bolt with an outdegree of zero (sinks), and the average outdegree of all vertices in the topology.

| Name | V | E | L | P | Src | Snk | AOD |
|---|---|---|---|---|---|---|---|
| Small | 10 | 17 | 4 | 0.40 | 3 | 3 | 1.70 |
| Medium | 50 | 88 | 5 | 0.08 | 17 | 17 | 1.76 |
| Large | 100 | 170 | 10 | 0.04 | 29 | 27 | 1.65 |

**Table 2.** The number of (V)ertices, (E)dges, and (L)ayers, the (P)robability to connect to vertices of different layers, the number of sources (Src) and sinks (Snk), as well as the average out-degree (AOD) of the vertices in the generated topologies.

In the basic configuration, all operators in the topologies were configured to use the same amount of computational resources and time. As real world topologies may not be balanced, we introduced a number of ways to create imbalance. We describe these modifications in the following paragraphs. Each modification will be motivated and its application described in detail. With all of them the goal is the same: we intend to

| Year | Description | # of Ops |
|---|---|---|
| 2003 | Data Dissemination Problem in [1] | 40 |
| 2004 | Linear Road Benchmark in [4] | 60 |
| 2013 | Linear Road Benchmark used in [12] | 7 |
| 2013 | DEBS'13 Grand Challenge Query[3] | 3 |

**Table 3.** Number of operators of topologies in literature.

generate multiple modified graphs from a base graph, which we can then optimize using Bayesian Optimization.

**Time Complexity** Each tuple takes $n$ units of compute resources (CPU cycles) to process. The amount of compute resources each tuple requires to be processed depends, naturally, on the task the processor has to achieve. We set a target value of 20 compute resource units per tuple in our experiments. As we need to simulate actual processing, we implemented a busy wait strategy in which we empirically set the complexity of the operations, such that 1 compute resource unit corresponds to about 1ms of execution time. Hence, the processing of one data tuple takes about 20ms on a system that is not overloaded. Others have reported values of up to 60ms [30] per tuple. In addition to the balanced base configuration we also generated imbalanced ones, where the required compute resource units vary across the topology. Specifically, we used a uniform distribution of compute length with a mean of 20 compute units (between 0 and 40), resulting in an average processing time of 20 in the whole topology.

**Resource Complexity** Bolts (or vertices) that are only constrained by CPU time are embarrassingly parallelizable and can be optimized soley by increasing their degree of parallelism. Other bolts may be constrained by resources that cannot be added by increasing their parallelism. If a task instance is slow because of a globally contentious resource, for example a central database, instantiating more tasks will not help improve the throughput and only waste resources on context switching. To simulate contentious resources, we flag a certain percentage of the processing time as being "resource contentious". This means that the time complexity of the respective bolts is multiplied with the total number of task instances for a given bolt to negate the effect of increasing parallelism for the affected bolt. To avoid unfair distribution of resource contention, this

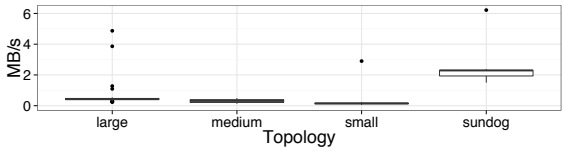



**Fig. 3.** Average network load in MB/s per worker for each topology.

percentage is based on the number of total compute resource units (see above in section 4.2), rather than just selecting a percentage of the bolts. For example, if we have a topology with 10 nodes which have an average time complexity of 20 and we want to have 25% contentious nodes, we select nodes with a total time complexity of 50 units of compute resources and flag them as 'contentious resources bolts'.

**Selectivity** For every incoming tuple, a task instance produces 0 to $n$ outgoing tuples. This is called the *selectivity* value of a bolt. Selectivity is not susceptible to the degree of parallelism. In contrast to processing time, the selectivity value not only influences the workload on downstream operators, it also incurs network traffic. However, in setups where the network is not the bottleneck, selectivity can be simulated using the value for time complexity: having a selectivity of more than 1 incurs increased workload on all downstream bolts in the topology. Hence, to simulate a higher selectivity value, we can as well increase the time value of all downstream nodes. Analogously, a selectivity value of less than 1 reduces the workload of all downstream nodes. For the experiments presented in this paper, we took care not to overload the network by using sufficiently large processing time values and omitted a special selectivity flag. Figure 3 shows the network utilization in megabytes per second (MB/s) as an average across all worker nodes in the cluster for all four types of topologies we used in our evaluations. Note that the network was not saturated in any of our experiments, as the cluster nodes are equipped with gigabit network cards that allow a theoretical upper limit of 128MB/s.

**Topology Generation** The topology generation for the synthetic topologies consists of (i) generating the base graphs using GGen, (ii) modifying the resulting graphs by randomly (but uniformly) changing the time complexity values and resource contention flags, and finally, (iii) generating

Storm topologies. The bolts in these topologies are linked using shuffle-grouping, meaning tuples are evenly shuffled among downstream bolts. This completes the description of the topology modifications. The concrete degree to which we applied these modifications will be described below in the section 4.2.

## 4.3 Cluster Configuration

This section describes the cluster hardware and software used for the experiments.

**Hardware** Many compute clusters that are in production in industry consist of several thousand commodity computers [2]. While we did not have a cluster of this magnitude at our disposal, we made an effort to simulate such a cluster by connecting the work station computers that our department offers to our students to work on, into an 80 machine Hadoop cluster. The student computers are iMac computers with Intel Core i5 CPUs (4 cores with each 2.7GHz), 8GB ram, and 250GB SSD hard drives. The iMacs are distributed over two rooms, in rows of at most 8 computers (some rows contain fewer computers). Each row is connected using a 1Gbps switches. All rows are connected over at most 2 Cisco Catalyst 4510R+E (48Gbps) switches. We scheduled our evaluations during off hours. However, we cannot exclude that there were students using the iMacs systems during the evaluations. We compensated for this by running each evaluation multiple times.

In version 0.23, Hadoop introduced support for other applications than MapReduce through its YARN[13] resource scheduler. For our experiments, we used the Storm-Yarn project,[14] which is an effort to run Storm inside a Hadoop cluster. In order to prevent the Hadoop cluster from going down because of a student accidentally shutting down his work station, we ran the Hadoop Job Tracker as well as the Zookeeper[15] instance on a separate machine. For this, we used a virtual machine with 4 simulated 2.6GHz CPUs.[16]

[13] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site
[14] https://github.com/yahoo/storm-yarn
[15] http://zookeeper.apache.org
[16] We use KVM as the virtualization plattform with storage on iSCSI.

**Software** All iMac computers ran OS X 10.9.5, having Java 1.8.0_11.jdk installed. The virtual machine running the job tracker ran on Debian 7.8 (wheezy). We used Hadoop 2.2.0 as the base system and Storm 0.9.2-incubating through Storm-Yarn 1.0-alpha orchestrated by Zookeeper 3.4.5.

# 5 Results

This section discusses the results of our evaluations. First, we compare throughput performance achieved when tuning parallelization. Second, we explore the practicality in terms of convergence speed of using Bayesian Optimization. Next, we investigate the robustness of our approach against problem size. We close with a discussion of the tuning of additional parameters.

## 5.1 Configuring Parallelism

In a first set of experiments, we were interested in finding out if the parallelism hints of a topology can effectively be chosen using Bayesian Optimization. We used Spearmint to choose a parallelism hint for each node in the topology and decide over the maximum number of task instances ("max-tasks") that Storm should instantiate. To ensure that the sum of tasks is smaller than max-tasks, we normalized the chosen hints using the max-task parameter. As a baseline we implemented a naive parallel-linear ascent (pla) optimizer, which sets the same parallelism hint on all spout/bolt nodes in the topology and increases them in parallel. We set the maximum number of evaluation runs to be 60. To prevent unnecessary evaluation runs for the pla strategies, we stopped the optimizer after measuring zero performance in three consecutive runs. As we possess detailed topological information for the synthetic topologies, we additionally created a set of experiments in which we leveraged the topological information. For these experiments we recursively calculated a "base parallelism weight" value for each node in the topology. For bolts, this base weight is equal to the sum of the weights of all their parent nodes. All spout nodes have a base weight of 1. The optimizer then only had to choose a multiplier for these base-parallelism weights. We denote optimizers working with this additional topological information with the letter "i" for "informed".

Figure 4 serves as an overview over the results from this comparison. We list results achieved using the bayesian optimizer (bo, we will discuss the bo180 values below), the parallel linear ascent optimizer (pla), the informed bayesian optimzier (ibo), and the informed parallel linear ascent optimizer (ipla). For each optimization step, we had the cluster process data for two minutes. Starting and stopping the topology took between 40 and 100 seconds. The duration of the optimization steps depends on the size of the topology and took between 13 and 518 seconds (see Section 5.3 below). We then ran the best configuration for each topology-optimizer combination 30 times. Given that our approach is probabilistic, we repeated the procedure and graphed the better of the two optimization passes in the figure, which shows the average of the 30 repetitions with the best configuration (error bars represent the minimum/maximum values).

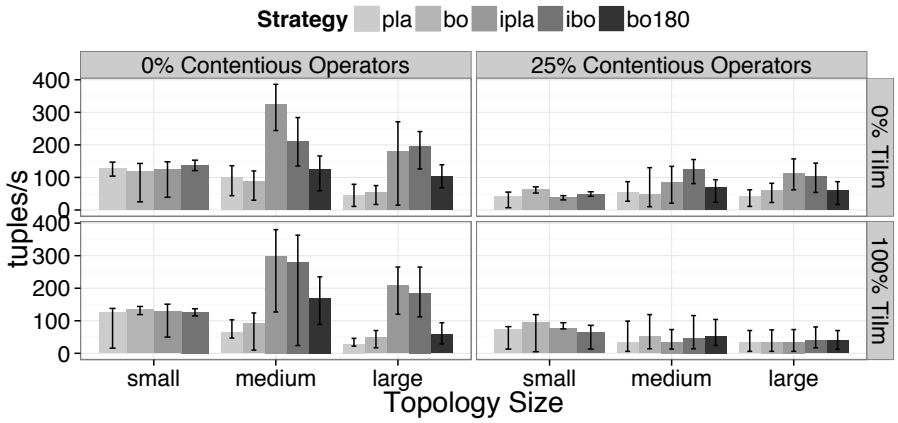



**Fig. 4.** *Throughput*: Average performance running synthetic topologies with varying time complexity imbalance and resource contention on an 80 node cluster (TiIm = time complexity imbalance, pla = parallel linear ascent, bo/bo180 = Bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

The top-left bar plot shows the results for topoloies for which the variance of time complexity is zero. In these homogenous topologies, each spout and bolt consumes the same number of CPU cycles to process a single incoming tuple. Also, we ignore resource contention. Under such conditions, setting all parallelism hints to the same value and increasing

them in parallel is a prudent optimization strategy. *ipla* dominates the field for medium and large topologies. The bayesian optimization strategies (*bo* and *ibo*) are unable to find a better configurations. For small topologies, all optimization strategies arrive at equally good solutions.

The lower-left bar plot in Figure 4 shows the results for the case in which the required CPU cycles to process tuples varies for each bolt. We observe that having topological information is still of use, however, Bayesian Optimization can partially compensate for the absence of such information (pla vs. bo) for medium and large topologies. For small topologies, all strategies arrive at equally good parallelism configurations.

In the upper-right plot of Figure 4 we experimented with the case in which temporal complexity is zero (e.g. homogenous bolts), however, we randomly selected 25% of the compute time to be dependent on "contentious resources" (see section 4.2). Essentially we bottlenecked 25% of all bolts. This experiment shows that topological information is still of value in such cases, however, Bayesian Optimization can help increase performance substantially for medium and large topologies.

In the lower-right corner of Figure 4, we finally tested the case in which we have both, heterogeneous time complexity, as well as 25% bottlenecked bolts. As we can see, topological information does not allow for any better configurations. In fact, for the large topologies all optimizers set values of or very close to 1 for all nodes the topology. The small topology configuration with time complexity imbalance and contentious resources, Bayesian Optimization without topological knowledge arrived at the best throughput results.

### 5.2 Convergence Speed

To assess the convergence speed we plotted the step at which we first measured the best performance for each experiment (Figure 5). As we ran each optimizer twice, we show minimum-maximum-average numbers over the two runs. Naturally, the bayesian optimizer needs many more steps than the linear parallel approach. Interestingly, having topological information, not only improved the overall result of the configuration, but also shortened the number of evaluation runs necessary, to arrive at this result. In four cases, the best configuration was only found in the 60st run. For this reason, we ran four configurations for 120 more

steps. The best result achieved in 180 steps is depicted in 4 as the *bo180* strategy. We observe that giving the bayesian optimizer more time to find good parallelism settings, yields better results in all cases. In Figure 6, we plotted the LOESS regression smoothing with span 0.75 for these experiments. The trendlines are consistent with the performance values in Figure 4: for the small and the medium topologies, good parallelim settings can be found within the first 50 and 100 optimization steps, respectively. For the large topologies, for which over 100 parameters need to be set, the setting with time imbalance (lower-left) seems to have benefitted most from the additional time and the trend line increases after 100 time steps.

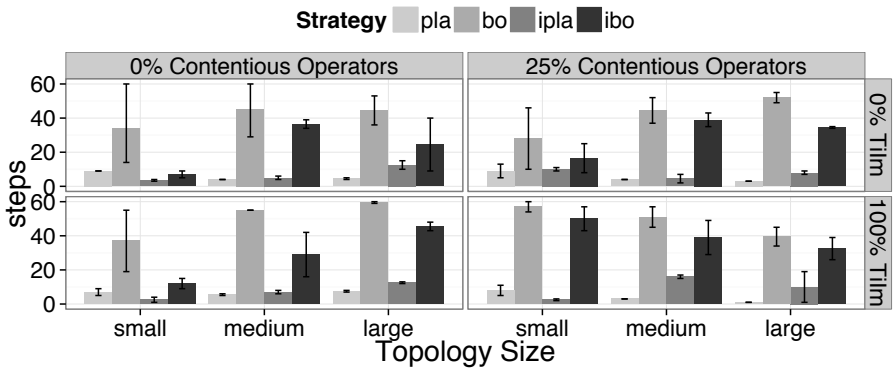



**Fig. 5.** *Convergence Speed*: Number of steps required to arrive at the maximum performance in terms of throughput for each experiment (TiIm = time complexity imbalance, pla = parallel linear ascent, bo = bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

## 5.3 Scalability

To assess the suitability of our approach for large parameter spaces, we measured the average optimizer run-time and plotted it in Figure 7. The pla and ipla times are barely visible, they lie all between 0 and 1 second. As we can see, the time required to choose the next configuration increases dramatically as we increase the topology size, and hence, the number of parameters to optimize. Spearmint needed an average of 35, 90, and 173 seconds for each optimization step for the small, medium, and large topologies (bo runs). Recalling that these topologies have 10,

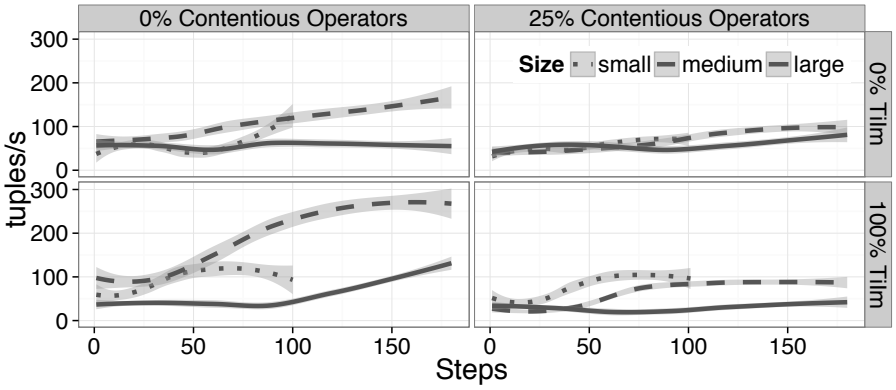



**Fig. 6.** Loess regression smoothing of the optimization steps of the bayesian optimizer setting parallelism hints.

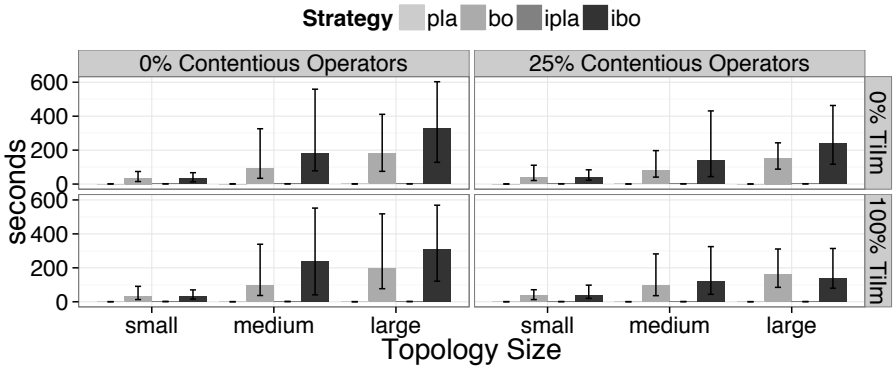



**Fig. 7.** *Scalability*: Average time elapsed between runs in seconds as a measure for how long one optimization step takes (TiIm = time complexity imbalance, pla = parallel linear ascent, bo = Bayesian Optimization, ipla = informed parallel linear ascent, ibo = informed Bayesian Optimization).

50, and 100 bolts, and hence, parallelism hints to optimize, we note that this increase is sublinear. The informed Bayesian Optimizer (ibo) required slightly more time with 36, 168, and 253 seconds per step, respectively. We assume this is due to the fact that we used floating points values for the weights as opposed to the simple integer values. These numbers increase also sublinearly. We observe increasing spreads between best and worst-case durations. However, even these numbers increase only sublinearly. Hence, all results indicate that the use of our approach is practical in terms of run-time.

To summarize these evaluations with synthetic data, we conclude that while Bayesian Optimization can be practically used to configure the parallelism of a distributed stream processor, it can only partially compensate for missing topological information. In situations, however, where this information is expensive to obtain or topologies are complex (e.g., due to joins or filters), Bayesian Optimization offers itself as a viable tool.

### 5.4 Optimizing Other Configuration Parameters

To assess the usefulness of Bayesian Optimization for configuring other aspects of a distributed stream processor for a real-world topology in combination with the degrees of parallelism of its operator nodes, we conducted an additional set of experiments, which we present in the following sections.

In these experiments, we used the Sundog topology presented in section 4.1. As we did not have topological information about the topology readily available (and they are non-trivial to derive), we only employed the parallel linear ascent (pla) and the bayesian optimizer (bo). We ran three different combinations of parameter sets: in a first set or experiments, we had the optimizers choose the parallelism hints as in our previous evaluations to get a baseline to compare to. Then, we created configurations for Spearmint to optimize parameter sets that include the parallelism hints along with the batch parallelism, batch size, and finally a set of concurrency related configuration parameters: the batch-size parameter lets us set the number of lines of text that Sundog ingests in one mini-batch. Batches can be processed in parallel. The parameter for batch-parallelism defines how many such batches can be in the processing pipeline concurrently. The last set of parameters that we included in the setup were all concurrency (cc) related parameters from Table 1: the number of worker and receiver threads, as well as the number of “acker” tasks that Storm should instantiate.

We present the results obtained from running these experiments in Figure 8. The best configuration of each optimizer was run 30 times. We present average throughput values in million tuples per second, denoting the maximum and minimum measured results with error bars.

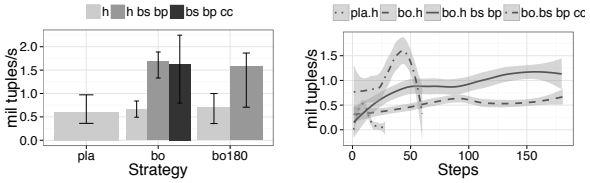
In a first comparison, and to get a baseline for later experiments, we looked at the performance that can be achieved by setting parallelism hint

(h) values. For these experiments, we used a batch-size of 50.000 lines and a batch-parallelism of 5, as these were the values used when Sundog was developed and manually tuned. As our cluster machines have 4 cores, we set a worker thread pool to 8. We did not set a value for the number of acker instances, so the default of one per worker host was used: 80 total in our case. We left the default value of 1 for the worker receiver thread count. Looking at the results in Figure 8a, we note that all three approaches (pla, bo, and bo180) achieve very similar average results (611k, 660k, and 699k tuples/s). A two-sided t-test deemed these differences statistically insignificant (p=0.05).

In a second set of experiments, we added the parameters for batch-parllelism (bp) and batch-size (bs) and had Spearmint choose values for these settings in addition to the parallelism hints resulting in substantial performance gains. We measured a throughput of 1.68 million tuples per second. This amounts to an improvement of 2.8x compared to the 611k tuples/second throughput measured when only optimizing the parallelism hints using pla. When looking at the parameter configurations we found that the bayesian optimizer changed the batch-parallelism from 5 to 16 and increased batch-size from 50.000 to 265.312 tuples. The Sundog developers reported that they never set these values that high, as the time it takes to process a batch of this size seemed unreasonably high.

In a last experiment, we explored if not spending the time on optimizing parallelism, but instead on fully concentrating on other parameters, would yield better performance. In this experiment, we fixed the parallelism hint for all bolts to the best value that the pla strategy yielded (11), and had Spearmint search the parameter space of all parameters listed in Table 1 except the parallelism hints. The result can be seen in Figure 8a (bs bp cc): even though the bayesian optimizer could spend 60 optimization steps on this reduced parameter space, the highest throughput measured in this experiment is comparable to the one achieved in the "h bs bp" cases. Indeed, two-sided t-tests revealed that the throughput of the "bs bp cc" run (1.63mio tuples/s) was not significantly different from the performance measured when searching the extended parameter space over 60 (1.68mio tuples/s) or 180 (1.58mio tuples/s) steps (p=0.05). Figure 8b shows the progress of the approaches: concentrating on only optimizing parallelism did not result in good performance even after 180 steps (dashed line). Configuring parallelism as well as batch size and batch parallelism (solid line), did yield good results, eventually.

The fastest way seems to be a combination of both approaches, where we first configured parallelism using the parallel-linear approach and enhanced the settings by optimizing the batch-size, batch-parallelism, as well as the number of threads used by the various subsystems (dot-dashed line).




(a) Throughput (b) Convergence

**Fig. 8.** Throughput and convergence speed for Sundog using parallel linear ascent (pla) and Bayesian Optimization (bo) to optimize the parallelism hints (h) with and without the batch size (bs) and the batch parallelism (bp), as well as a set of concurrency (cc) related parameters.

# 6 Conclusions and Future Work

We presented and evaluated an approach for the configuration of distributed stream processors. We implemented the approach using a set of synthetic and real-world Storm topologies. We had a bayesian optimization framework find optimal parameter settings to achieve high throughput and compared against a parallel linear optimization approach. Our results suggest that our approach is viable and can find parameter configurations that lead to substantial throughput improvements by a factor of up to 2.8 in the best case.

There are some limitations to our work. First, Bayesian Optimization using Gaussian Processes assumes that the objective function is continuous. This may not always be the case when configuring the parallelism of a distributed stream processor. To what extent this negatively influenced the results in our auto-parallelization experiments is subject to future work. Second, as even small sample differences influence the decision process of the bayesian optimizer, our setup could be improved by running each sampling run multiple times and by using the average performance for each tested parameter configuration.

We believe that Bayesian Optimization is a viable tool for the field of distributed computing. Especially for tuning systems with a large configuration parameter space in which the impact of every single parameter cannot easily be predicted. As such, we are convinced that our work is of interest to the community.

## Acknowledgments

We would like to thank Hanspeter Kunz and Enrico Solca for the many hours they spent helping us setup and run the cluster we used for the evaluations.

# References


[1] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal The International Journal on Very Large Data Bases*, 12(2):120–139, aug 2003. doi: 10.1007/s00778-003-0095-z. URL http://dx.doi.org/10.1007/s00778-003-0095-z.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*. Association for Computing Machinery (ACM), 2008. doi: 10.1145/1402958.1402967. URL http://dx.doi.org/10.1145/1402958.1402967.

[3] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems - DEBS '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2488222.2488267. URL http://dx.doi.org/10.1145/2488222.2488267.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, VLDB '04, pages 480–491. VLDB Endowment, 2004. URL http://dl.acm.org/citation.cfm?id=1316732.

[5] James Bergstra, D Yamins, and D Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *Proceedings of the 30th International Conference on Machine Learning*, 2013. URL http://jmlr.org/proceedings/papers/v28/bergstra13.html.

[6] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning.

*CoRR*, abs/1012.2599, 2010. URL http://arxiv.org/abs/1012.2599.

[7] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel. A cost-based approach to adaptive resource management in data stream systems. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):230–245, feb 2008. doi: 10.1109/tkde.2007.190686. URL http://dx.doi.org/10.1109/TKDE.2007.190686.

[8] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (ICST), 2010. doi: 10.4108/icst.simutools 2010.8667. URL http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8667.

[9] Michael Daum, Frank Lauterwald, Philipp Baumgärtel, Niko Pollner, and Klaus Meyer-Wegener. Black-box determination of cost models' parameters for federated stream-processing systems. In *Proceedings of the 15th Symposium on International Database Engineering & Applications - IDEAS '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/2076623.2076654. URL http://dx.doi.org/10.1145/2076623.2076654.

[10] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1): 1–140, 2006. doi: 10.1561/1900000001. URL http://dl.acm.org/citation.cfm?id=1331940.

[11] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, and Jasper Snoek. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *Advances in Neural Information Processing Systems Workshop on Bayesian Optimization in Theory and Practice*, 2013.

[12] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2463676.2465282. URL http://dx.doi.org/10.1145/2463676.2465282.

[13] Lorenz Fischer, Roi Blanco, Peter Mika, and Abraham Bernstein. Timely semantics: A study of a stream-based ranking system for entity relationships. In *Proceesings of the 14th International Semantic Web Conference - ISWC '15*, 2015.

[14] Buğra Gedik, Henrique Andrade, and Kun-Lung Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proceeding of the 18th ACM conference on Information and knowledge management - CIKM '09*. Association for Computing Machinery (ACM), 2009. doi: 10.1145/1645953.1646061. URL http://dx.doi.org/10.1145/1645953.1646061.

[15] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, jun 2014. doi: 10.1109/tpds.2013.295. URL http://dx.doi.org/10.1109/TPDS.2013.295.

[16] Joseph Gomes and Hyeong-Ah Choi. Cost-based solution for optimizing multi-join queries over distributed streaming sensor data. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Institute of Electrical & Electronics Engineers (IEEE), nov 2006. doi: 10.1109/colcom.2006.361871. URL http://dx.doi.org/10.1109/COLCOM.2006.361871.

[17] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward bound: Planning for beneficial migration of enterprise applications to the cloud. In *Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM - SIGCOMM '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1851182.1851212. URL http://dx.doi.org/10.1145/1851182.1851212.

[18] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems - DEBS '14*. Association for Computing Machinery (ACM), 2014. doi: 10.1145/2611286.2611294. URL http://dx.doi.org/10.1145/2611286.2611294.

[19] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-25566-3_40. URL http://dx.doi.org/10.1007/978-3-642-25566-3_40.

[20] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, dec 2000. doi: 10.1145/371578.371598. URL http://dx.doi.org/10.1145/371578.371598.

[21] Jonas Mockus. On bayesian methods for seeking the extremum and their application. In *IFIP Congress*, pages 195–200, 1977.

[22] Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 2(117-129), 1978.

[23] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *The 28th International Conference on Distributed Computing Systems*. Institute of Electrical & Electronics Engineers (IEEE), jun 2008. doi: 10.1109/icdcs.2008.11. URL http://dx.doi.org/10.1109/ICDCS.2008.11.

[24] Sven Schmidt. *Quality-of-Service-Aware Data Stream Processing*. PhD thesis, Technischen Universität Dresden, 2007.

[25] Scott Schneider, Henrique Andrade, Buğra Gedik, Alain Biem, and Kun-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. Institute of Electrical & Electronics Engineers (IEEE), may 2009. doi: 10.1109/ipdps.2009.5161036. URL http://dx.doi.org/10.1109/IPDPS.2009.5161036.

[26] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2370816.2370826. URL http://dx.doi.org/10.1145/2370816.2370826.

[27] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems - NIPS '12*, 2012. URL http://arxiv.org/abs/1206.2944.

[28] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-WEKA. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2487575.2487629. URL http://dx.doi.org/10.1145/2487575.2487629.

[29] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394, 2002. doi: 10.1002/jos.116. URL http://dx.doi.org/10.1002/jos.116.

[30] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems - DEBS '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2335484.2335515. URL http://dx.doi.org/10.1145/2335484.2335515.

[31] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219, jun 2007. doi: 10.1145/1272998.1273020. URL http://dx.doi.org/10.1145/1272998.1273020.

# Timely Semantics:
## A Study of a Stream-based Ranking System for Entity Relationships

This chapter is based on a paper that has been accepted at and published in the proceedings of the *14th International Semantic Web Conference (ISWC 2015)*.

# Timely Semantics:
## A Study of a Stream-based Ranking System for Entity Relationships

Lorenz Fischer[1], Roi Blanco[2], Peter Mika[2], and Abraham Bernstein[1]

[1] University of Zurich, Department of Informatics, Zurich, Switzerland
{lfischer, bernstein}@ifi.uzh.ch
[2] Yahoo Labs, London, United Kingdom
{roi, pmika}@yahoo-inc.com

**Abstract.** In recent years, search engines have started presenting semantically relevant entity information together with document search results. Entity ranking systems are used to compute recommendations for related entities that a user might also be interested to explore. Typically, this is done by ranking relationships between entities in a semantic knowledge graph using signals found in a data source as well as type annotations on the nodes and links of the graph. However, the process of producing these rankings can take a substantial amount of time. As a result, entity ranking systems typically lag behind real-world events and present relevant entities with outdated relationships to the search term or even outdated entities that should be replaced with more recent relations or entities.
This paper presents a study using a real-world stream-processing based implementation of an entity ranking system, to understand the effect of data timeliness on entity rankings. We describe the system and the data it processes in detail. Using a longitudinal case-study, we demonstrate (i) that low-latency, large-scale entity relationship ranking is feasible using moderate resources and (ii) that stream-based entity ranking improves the freshness of related entities while maintaining relevance.

## 1 Introduction

In the past years, one of the major developments in the evolution of search engines has been the move from serving only document results to providing entity-based experiences. In contrast to the document results that are crawled from the Web, these experiences are typically built on top of a knowledge base assembled by the search engine provider from various sources of general and domain knowledge. All three major US search engines (Bing, Google, and Yahoo) have developed features that make use of such a knowledge base, and in particular to provide large information boxes which, at the time of writing, appear on the rightmost column of the interface for all three search engines. In all three cases, the displays also provide recommendations for related entities that the user may also want to explore.

Knowledge bases are typically organized in the form of an entity-relationship graph with additional facts attached to the entities and relationships. While the facts represented in the graph rarely change, the timeliness of relationships can be significantly impacted by real world events. For example, in the domain of entertainment, a movie release could drive significant interest towards the collaborations of actors, or news of an impending celebrity divorce may raise interest into a couple. Similarly, in the domain of sports, a game could drive searches toward the players that participated in certain actions during the game, etc. The features that are used for measuring the importance of these relationships thus also need to be reassessed as a result of these events.

Entity recommender systems typically work by exploiting query logs for predicting the relevance of a related entity, as query logs provide an accurate reflection of current interests. Traditionally, such logs are collected and processed using offline, distributed batch processing systems such as Hadoop MapReduce.[3] These systems are designed to handle large volumes of data but at the cost of significant processing latency. More recently, a new class of distributed systems based on stream processing have become available, opening up the potential for new or improved applications of semantic technologies.

In this work, we describe *Sundog*, a stream processing based implementation of an entity-recommender system and show that by exploiting the temporal nature of search log data, we are able to significantly improve the quality of recommendations compared to static models of relevance, in particular with respect to freshness. The architecture of Sundog is based on a system that has previously been presented at ISWC – Spark [2]. To understand the differences in technology, we provide a comparison to the architecture of the batch-processing based predecessor. We then describe a longitudinal study that evaluates the relevance and freshness of the results computed by the system over a number of consecutive days, using different window sizes and temporal lag in computing the model. We show the benefits of using increasing amounts of data and reducing the lag in processing, namely a relevance and *freshness* increase of over 24% with respect to approaches that use stale data, in the best case. We conclude by discussing improvements and other potential applications of our work.

[3] http://hadoop.apache.org

## 2 Related Work

Our *Sundog* system is an entity ranking system facilitating semantic search through the application of supervised machine learning techniques to features extracted from query log data. Hence, this section succinctly reviews the related work on (i) semantic search & entity ranking and (ii) temporal information retrieval.

With the introduction of entity-based experiences such as infoboxes, direct answers and *active objects* [16], the disambiguation of query intent and search results have gained in importance, because in these applications mistakes in query interpretation are immediately obvious to the user. However, the semantic gap between the words in user query and the descriptions of entities in the entity-graph can be significant [18]. Entity ranking, or ad-hoc object retrieval is aimed at finding the most relevant entity related to the user's query, and it has been the focus of many recent studies [2, 14, 20, 19, 24]. Pound et al. provide a query classification of entity-related search queries and define evaluation metrics for the entity retrieval task [20]. This task has also been the focus of evaluations in TREC [1] and other venues such as the SemSearch challenges [3]. A variant of the ad-hoc object retrieval task is the recommendation of related entities, where the focus is on ranking the relationships between a query entity and other entities in the graph, see Kang et al. [14] and van Zwol et al. [24]. More recently, Blanco et al. [2] present their work on the *Spark* system, which is a continuation of the work of Kang et al.

Temporal aspects have gained traction in information retrieval (IR) over the last couple of years and have found applications in document ranking [7, 6, 9], query completion [22], query understanding [15, 8, 17], and recommender systems [26, 5, 21]. Shokouhi et al. [22] analyse temporal trends and also use forecasted frequencies to suggest candidates for auto completion in web search. Kulkarni et al. analyse different features to describe changes in query popularity over time, to understand the intent of queries [15]. In [7], Dai et al. use temporal characteristics of queries to improve ranking web results using machine learned models. They use temporal criteria for their page authority estimation algorithms in [6]. More specifically, they propose a temporal link-based ranking scheme, which also incorporates features from historical author activities. Dong et al. identify *breaking news queries* by training a learning to rank model with temporal features extracted from a page index such

as the time stamp of when the page was created, last updated, or linked to [8]. Elsas et al. analyzed the temporal dynamics of content changes in order to rank documents for navigational queries [10]. More related to the topic of query intent analysis, Metzler et al. [17] propose to analyse query logs in order to find base queries that are normally qualified by a year, in order to improve search results for *implicit year qualified* queries. The work that is probably most closely related to our study is by Dong et al. [9]. The authors use realtime data of the micro-blogging website Twitter to extract recency information and train learning to rank models, which in turn are used to rank documents in web search. The recency information from Twitter was then successfully used to rank documents, which promotes documents that are both more fresh and more relevant.

## 3 System Description

The entity ranking system employed at Yahoo – Spark – is implemented as a batch-processing based pipeline. For this study, we present Sundog, which implements parts of the Spark pipeline using a distributed stream processing framework. A full system description of the production system is beyond the scope of this paper and we refer to [2, 23] for details. However, for the sake of reproducibility and to understand the various design decisions made when building the system for our experiments, it is necessary to have an understanding of Spark. For this reason, we are first going to introduce the most important parts of the Spark processing pipeline, before we elaborate in detail, how and in what aspects Sundog is different from the original system. We then describe the various performance optimizations we applied. We end the section with performance statistics of the system.

### 3.1 The Spark Processing Pipeline

Figure 1 gives a high level overview over the Spark ranking pipeline. The ranking essentially happens in three steps: Volatile data sources are used to generate co-occurrence features of entity pairs that are part of the relationships found in a semantic knowledge base (1). Data sources used for this step are Yahoo search logs, tweets from Twitter, and Flickr image tags. Note that for Sundog, we limited ourselves to only use search logs as input data. Next to features extracted from these volatile sources,

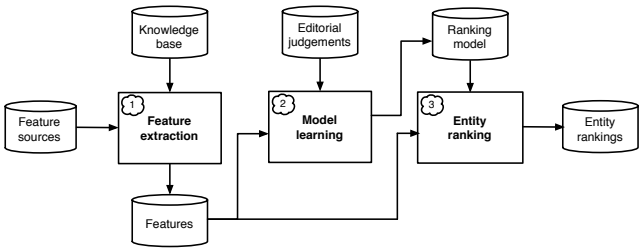



**Fig. 1.** High-level architecture of the Spark entity ranking system.

semantic information such as the entity types and relationship types are leveraged as features. The next step involves the training of a decision tree model using editorial judgements that have previously been collected for a limited set of entity pairs (2). The resulting ranking model is then used to generate entity rankings for all the entity pairs for which features were extracted (3). Disambiguation is conducted in a post-processing step and it is based on a popularity measure derived from Wikipedia. For more information on pre- and post-processing as well as the serving facility, we refer to [2].

**Model Learning & Ranking** Spark employs learning to rank approaches in order to derive an efficient ranking function for entities related to a query entity.

Formally speaking, the goal of the Spark ranking system is to learn a function $h(\cdot)$ that generates a score for an input query $q_i$ and an entity $e_j$ that belongs to the set of entities related to the query $e_j \in \mathcal{E}^{q_i}$. Together, $q_i$ and $e_j$ are represented as a *feature vector* $\mathbf{w_{ij}}$ that contains one entry per feature extracted. The input of the learning process consists of *training data* of the form $\{T(q_i) = \{\mathbf{w_{ij}}, l_{ij}\}\}_{q_i \in \mathcal{Q}}$, where $l_{ij} \in L$ is a manually assigned label from a pre-defined set. Spark uses a 5-level label scale ($l \in \{$*Bad, Fair, Good, Perfect, Excellent*$\}$) and the assignment from examples $(q_i, e_j)$ was done manually by professional editors, according to a pre-defined set of judging guidelines. The query set $\mathcal{Q}$ is comprised of editorially picked entities and random samples from query logs. This is expected to mimic the actual entity and query distribution of the live system. The training set might also contain *preference* data, that is, labels that indicate that an entity is preferred over another one for a particular query. The ranking function has to satisfy the set of preferences as much as possible and at the same time is has to *match* the label in the

sense that a particular loss function is minimized, for instance square loss $\frac{1}{|Q|}\sum_{q_i \in Q}\frac{1}{|E^{q_i}|}\sum_{e_j \in E^{q_i}}(l_{ij} - h(\mathbf{w_{ij}}))^2$, for a set of test examples.

Similarly to [23], Spark uses Stochastic Gradient Boosted Decision Trees (GBDT) for ranking entities to queries [13, 12]. GBRank is a variant of GBDT that is able to incorporate both label information and pairwise preference information into the learning process [27] and is the function of choice we adopted for ranking in Spark. The system was trained using ∼30K editorially labelled pairs and ten fold cross-validation. Each time a model is learned the system sweeps over a number of parameters (learning rate, number of trees and nodes, etc.) and decides on their final value by optimizing for *normalized discounted cumulative gain* (NDCG).

The features included in the system comprise a mixture of *co-occurrence features* and *graph-based features*. Co-occurrence features compute several statistics on mentions of pairs of entities appearing together in the data sources (conditional and joint probabilities, KullbackâĂŞLeibler divergence, mutual entropy, etc.). Other features include the types of entities and types of their relationships. In contrast to Spark, Sundog does not currently include graph-based features such as PageRank or the number of shared vertices (common neighbors) between two entities. It does, however, create features using various linear combinations of the features mentioned before as well as make use of the semantic features (type annotations). For a detailed description of these features we refer to [2].

### 3.2 The Sundog System

In this section we present the implementation details of the Sundog system. First, we describe the programming framework used to build the system. Next, we describe Sundog itself, before presenting a series of optimizations that were implemented to achieve the necessary performance.

**Storm & Trident** Sundog was implemented using the *Storm*[4] realtime computation framework. Storm is best described as the Hadoop MapReduce for stream processing. Similar to MapReduce, data is partitioned,

[4] http://storm-project.net

distributed amongst, and processed by multiple compute nodes concurrently. A Storm application—a *topology*—is a directed graph consisting of spout and bolt nodes.

*Trident* is a higher level programming framework that is part of the Storm distribution. Trident offers higher level concepts such as aggregates, joins, merges, state queries, functions, filters, and methods for state handling. As the Sundog system relies heavily on the computation of state in the form of feature statistics that have to be computed continuously, we chose to use *Trident* for the implementation. In Trident, tuples are processed and accounted in mini-batches, offering consistency guarantees on a per-batch basis.

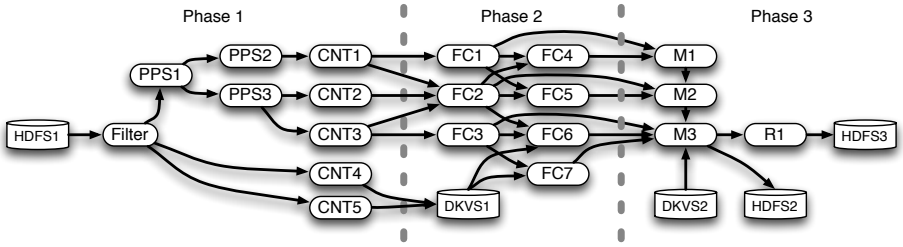



**Fig. 2.** The *Sundog* topology.

**Topology Design** The topology of Sundog can be roughly divided into three phases as depicted in Figure 2. Please note that for the sake of clarity, we show less nodes in the schematic depiction than the actual topology has. Each phase has to fully finish processing, before the next phase can start. For example, before we can compute the probability features of one mini-batch in phase 2, we first have compute the counter values for that mini-batch in phase 1. In the following paragraphs we are going to explain each of these phases in more detail.

In the first phase, query log data is read from the *Hadoop Distributed Filesystem* (HDFS). For the sake of reproducibility of our experiments and because we needed to be able to process historic log data, we chose to read from HDFS, rather than reading from a volatile message queue. We then filter out all search queries that do not contain at least one known entity name. The basis for this filtering is the same knowledge base (KB) that is used in the Spark system. For the experiments in this paper, we relied on the KB that was in production when we ran our experiments.

From this reduced data stream, we already count certain events such as the number of distinct users or the number of search events. For other counters, we first have to build entity pairs from the query terms in a series of preprocessing steps (PPS1-3). We count the number of search events and unique users overall (for each entity pair and for each entity). As most of these counters count events and unique users per entity or entity pairs, the relevant data can readily be partitioned and the computation therefore run in parallel on multiple compute nodes. Some counters, however, have a global character and their value has to be aggregated across all data partitions. Their values are stored in an external distributed key-value store (DKVS1) to enable access from all compute tasks in later phases of the processing pipeline. The second phase consists of computing the actual feature values from the counter values (FC1-7). In the final phase (phase 3), the computed features are merged together and complemented with semi-static features that are read from a table in the distributed key-value store (DKVS2 in Figure 2). Semi-static features are features that do not change often, or not at all. For example the semantic type of an entity or a relationship between two entities is assumed to be mostly static. After all features have been merged together, a score is computed for each entity pair using a GBDT model. Both, the feature values and the final scores are written back to HDFS (HDFS2 and HDFS3 in Figure 2).

**Optimizations** A major difference between a Storm topology and a MapReduce-based data pipeline is, that while between every MapReduce job, data necessarily needs to be written to and read from disk, Storm does not have this requirement. Tuples can pass between bolt instances without ever being written to disk. As long as there is enough memory, state can be kept in memory for fast access. In its current implementation, Sundog only stores two counter values persistently in an external storage. All other information is kept in memory. For this reason, we implemented several optimizations that reduce the memory footprint and the network traffic incurred by the system. In this section, we list the most important ones.

**HyperLogLog** For some of the features, we need the number of unique users that searched for a given entity or a pair of entities. These counts are necessary to reduce the impact any single user can have on the ranking of an entity. For example, a fan of a football team may search for the name

of his team very often together with the names of several other related entities. Normalizing with the number of unique users (rather than the number of search events) reduces the impact that any single search user has on the ranking.

The naïve way of counting uniques is to create a hash-set for each entity and entity pair – the values we want to count uniques for. For every search event, we could then add the user identifier to the hash-sets of the corresponding entity or entity pair. As hash-sets prevent duplicates from being stored, the size of the hash-set automatically represents the number of unique users that searched for a given entity or entity pair. The disadvantage of this method is, that we need one hash-set for every entity and entity pair and store the user identifiers of each user who searched for the given entity/entity pair. This has a worst case space complexity of $(e + ep) * u$, where $e$ is the number of entities, $ep$ is the number of entity pairs, and $u$ is the number of users. With millions of users and millions of entity pairs, this number can become prohibitively large. To circumvent this, we used an implementation of the *HyperLogLog* algorithm proposed by Flajolet et al. [11]. More specifically, we used the *stream-lib*[5] library for approximate counting. The fact that approximate counting - or cardinality estimation - does not provide us with exact counts, should not have a great impact on our results, as we normalize all values with the same "imprecise" counts for unique users. For the experiments presented in this paper, we chose a relative standard deviation of 1% as the target accuracy for the *HyperLogLog* estimator.

**Dictionary Encoding & Bloom Filters** As we only have to work with a limited set of entity and relationship types, we use dictionaries to encode both of these values. The compressed dictionary file is so small, that we were able to include it in the jar file deployed on the servers to make it available on all machines.

As described in Section 3.2 the information about semi-static features is stored in a distributed key-value store. The data in this database is also used to filter invalid entity pairs early in the processing pipeline. This ensures that we do not compute feature values for entity pairs that eventually turn out to be invalid. For example the pair *"Brad Pitt - Zurich, Switzerland"* may be found in the search logs, because a user searched

[5] https://github.com/addthis/stream-lib

for these two entities in the same search query. However, the KB may not contain a relationship entry for the two entities *"Brad Pitt"* and *"Zurich, Switzerland"*. In fact, the vast majority of potential entity pairs are invalid as there are certain geographical locations that have names that (after the text normalization process) are lexicographically equivalent to some words in the English dictionary. For example, there are several villages in Norway with the name *"Å"*. This leads to many candidate entity pairs between "a" and other entities that have no semantic relationship with each other. In order to filter these, we need to check against the KB. To reduce the number of requests to the KB, and hence the number of network requests in the process, we use Bloom filters [4]. For the experiments presented in this paper, we added all entity relations in the KB into a Bloom filter. The resulting data structure is included in the application deployment.

### 3.3 Runtime Characteristics & Performance

There are many factors that influence the performance of a distributed system. For the sake of reproducibility, it may be of interest to the reader to learn more about the setup of our cluster and the configuration parameters that we used for the evaluation of Sundog. For this reason, we present the setup of our system in this section by first describing the general setup of the Storm cluster, before giving some insight into how we configured our topology in order to get better performance out of the hardware that we were able to use.

**Cluster Configuration** We ran our evaluations on a cluster of machines that are connected to a 1 Gbit/s network, each having 24 2.2GHz cores and 96GB of RAM. The service that starts and stops Storm worker instances is called the *supervisor* service. There is one supervisor per machine in the cluster. All supervisors are centrally managed by a master server, the *nimbus* node. Communication between the nimbus service and the supervisors happens over a *Zookeeper*[6] cluster. The *nimbus* server schedules work among the available supervisor nodes. For our experiments, we were given exclusive access to 40 machines. We configured *Storm* to start 8 worker instances (JVMs) per supervisor, each having 12GB of RAM.

[6] http://zookeeper.apache.org

**Job Configuration & Performance** Table 1 lists setup parameters we used to configure Storm and Trident: We ran our experiments on 40 supervisors, each having 8 workers which resulted in 320 workers in total. These workers were executing 2449 task instances, of which 40 were spout instances and 2087 were regular bolt instances. The remaining bolt instances are acker-instances or Trident coordinator bolts. We ran one spout instance for each machine. Each of these read and emitted 100,000 log lines per batch. One batch took on average 40 seconds to complete, which means that the system ingested about 100,000 log lines per second. We found that neither increasing nor decreasing the batch-size led to increased throughput. Most likely a result of the bookkeeping overhead of Storm becoming proportionally more expensive with smaller batches, while larger batches just increased the processing time per batch. The system transferred about about 2.5 million messages per second within the topology. As running multiple batches concurrently did not yield higher throughput and only increased the chances for batches to time out, we always only processed one batch at a time. This led to the situation that we barely used all of the available resources, because, as mentioned in Section 3.2, certain parts of the computation need to wait for other parts to complete. This suggests that there may be further potential improvements in terms of resource utilization.

| **Parameter** | **Value** |
|---|---|
| Workers | 320 |
| Spouts (Spout Tasks) | 1 (40) |
| Bolts (Bolt Tasks) | 30 (2087) |
| Total Task Instances | 2449 |
| Concurrent Batches | 1 |
| Batch Size (log lines) | 100'000 |
| Average Batch Time (Seconds) | $\approx 40$ |
| Log Lines per Second | 100'000 |
| Transferred Messages per Second | 2.5 mio |

**Table 1.** Sundog configuration parameters and performance numbers of a typical evaluation run.

While the underlying platforms of Sundog and Spark are vastly different and the performance indicators can therefore not easily be compared directly, it is interesting to note, that even though Spark is running on a cluster that has two orders of magnitude more machines, Sundog is

still able to process comparable amounts of data in about $\frac{3}{4}$ of the time used by Spark.

# 4 Evaluation

Sundog allows us to compute feature values and entity rankings in much less time compared to the old Spark system. This in turns enables us to use more recently collected data for the ranking process. Hence, we are interested in three things: First, we investigate the impact of data recency on the entity rankings. For this we are interested in measuring the quality of the rankings in terms of freshness and relevance. Secondly, we analyze if fresh rankings are more useful to users. Lastly, we evaluate how the amount and the age of data used to train the system impact performance.

## 4.1 Experimental Setup

We evaluated the rankings that Sundog produces on four different days over the course of a week. We had human editors assess the rankings with regards to relevance and freshness on each day. In this section, we present the experimental setup of our evaluation. First, we describe the data sets that we collected, before we describe in detail how our editors assessed the generated rankings.

**The Data** For our experiments, we produced entity rankings using search log data of three time periods. For each time period we grouped the log files in sets of different sizes (windows). Each log file set $s_i \in S$ has a window of size $w_i$ and an end date $d_i$. The window size is inclusive. Hence, for a set $s_i$ with a window size $w_i = 7$ and end date $d_i = 2014/01/12$, the respective start date is defined as $d_i - w_i + 1 = 2014/01/06$. We differentiate between three different time periods or epochs, so three collections of *old*, *recent*, and *new* sets of log files. As we ran our experiments on 4 different days, the values for the *new* epoch changed. The end date for the *new* epoch is defined as:

$$d_n \in \{2014/01/20, 2014/01/21, 2014/01/22, 2014/01/23\}$$

For the *recent* and the *old* epoch we chose $d_r = 2014/01/12$ and $d_o = 2013/12/31$, respectively, to simulate the situation in which the rankings would be used during a period of two to three weeks. For each period we

compiled a data set of three different sizes $w \in \{1, 7, 30\}$. Table 2 lists the resulting data sets.

| Epoch | Window | Dates |
|---|---|---|
| Old | 1 Day | 2013/12/31 |
| | 7 Day | 2013/12/25 – 2013/12/31 |
| | 30 Days | 2013/12/2 – 2013/12/31 |
| Recent | 1 Day | 2014/1/12 |
| | 7 Day | 2014/1/6 – 2014/1/12 |
| | 30 Days | 2013/12/14 – 2014/1/12 |
| New | 1 Day | 2014/1/20...23 |
| | 7 Day | 2014/1/14...17 – 2014/1/20...23 |
| | 30 Days | 2013/12/22...25 – 2014/1/20...23 |

**Table 2.** Data sets collected for the evaluation.

For each of the data sets we first had Sundog generate the feature values which are stored in files. We then used these feature files to train *Gradient Boosted Decision Tree* (GBDT) models (see 3.1 for details). The resulting models were then used to generate the entity rankings, again stored in files. For each feature file and its corresponding model, we generated one ranking file. In addition, to test the performance of a model that has been generated with an old feature file on freshly generated feature values, we also generated some ranking files using models trained on old data and feature files extracted from new search log data. Note that for all rankings where we used models trained with historic data, we only scored the feature files on models of the corresponding window size, as the feature values in the feature files would otherwise be incompatible with the models.

The resulting ranking files contained a ranking score for each entity pair that at least one user searched for within the corresponding time window. As the number of such rankings can be quite large, we restricted ourselves to evaluate only a subset of all pairs. As we are mostly interested in evaluating for freshness, we selected the top-60 of all queries that matched the label of entities in the KB. This ensured, (i) that we only select queries for which related entities would actually be shown on the search page, and, as the entities in question were "trending", (ii) increased the likelihood that recency would be a factor for the relationships. We then took all entity pairs that we could find from all 15 ranking files for that day and pooled the query-entity pairs. With at most 10 related entities on the result page, this yielded a pool of at most $60 \times 15 = 900$ entity pairs per day - which was the upper limit of entity pairs that our

human editors could evaluate in respect to relevance and freshness in a day. Table 4 lists the exact numbers of query-pairs for each day.

**Editorial Judgement** We asked a group of expert search editors working for Yahoo to judge entity pairs in terms of relevance and freshness. Table 3 lists the categories from which the editors could select. The editors were trained and instructed to judge each relationship from the viewpoint of "today". We asked the editors to research the relationships which they did not know about, in order to provide a well founded judgment.

| Recency Categories | |
|---|---|
| Super Recent | Is current today or yesterday |
| Very Recent | Was current the past week |
| Recent | Relevant in the past year |
| Reasonable | A bit old, but still popular |
| Outdated | There are better connections |
| NA or NJ | Freshness is not a factor |
| **Relevance Categories** | |
| Super Related | Most interesting factual relationship |
| Closely Related | Related in a meaningful or useful way |
| Mostly Related | A little off, but makes sense |
| Somewhat Related | Not a meaningful or useful suggestion |
| Embarrassing | Does not make sense |
| N/J | No judgement possible |

**Table 3.** Available recency and relevance categories and their description.

| Date | Pairs | Super-Recent | Super-Related |
|---|---|---|---|
| 2014/01/20 | 819 | 57 | 290 |
| 2014/01/21 | 696 | 34 | 184 |
| 2014/01/22 | 865 | 54 | 171 |
| 2014/01/23 | 785 | 34 | 196 |

**Table 4.** The number of query pairs evaluated on each day with the corresponding number of pairs that were judged *Super-Recent* or *Super-Related*, respectively.

We measure the performance on both *relevance* and *freshness* using standard metrics such as normalized discounted cumulative gain (NDCG), precision, and mean average precision (MAP). Given that we are considering freshness as a discrete, graded variable, we report on the same metrics as relevance, but using the editorial labels for recency.

### 4.2 Results & Discussion

**Fresh Data Is Better** In Figure 3 we present our findings on the impact of data freshness on relevance and freshness scores: In the two charts in the upper row we plotted the NDCG scores using the top-10 and the top-5 results. We chose top-10, because Spark always shows the top-10 ranked related entities. Sundog may not be able to always find 10 related entities. This has several reasons: Firstly, Sundog only uses one of the four data sources that Spark uses. Secondly, while Spark uses default values for all features and entity pairs that could not be found in the data, Sundog only computes features values, and hence rankings, for entity pairs that we were able to find in the data. As we are currently mostly interested in freshness of relationships it makes sense not to include relationships that were not of any importance to our users during the time we collected the data. For this reason, Figure 3 also shows the NDCG values for the top-5 ranked entities.

It is apparent, that for sufficiently large time windows, *new* data always produces entity-rankings that are both fresher and also more relevant in general. If the data only contains log data from a single day, we see that while the data that was collected most recently still consistently produces superior rankings, the difference between the rankings of the *recent* data compared to the *old* data is negative. Looking at the numbers in Table 5a we can see a similar picture: Using window sizes of 7 and more days, we observe a significant improvement in terms of freshness when using more recent data.

| | Old | Recent | New |
|---|---|---|---|
| 1 | 0.3600 | 0.3446 (-4.20%) | $0.3868^{*}$ (+11.13%) |
| 7 | 0.3945 | $0.4317^{*}$ (+9.44%) | $0.4870^{*\dagger}$ (+24.38%) |
| 30 | 0.4569 | $0.4994^{*}$ (+9.30%) | $0.5335^{*\dagger}$ (+16.75%) |

(a) Freshness

| | Old | Recent | New |
|---|---|---|---|
| 1 | 0.4499 | 0.4522 (+0.66%) | $0.4958^{\dagger}$ (+10.20%) |
| 7 | 0.5107 | $0.5913^{*}$ (+15.80%) | $0.6123^{*}$ (+19.90%) |
| 30 | 0.6041 | $0.6588^{*}$ (+7.71%) | $0.6589^{*}$ (+9.05%) |

(b) Relevance

**Table 5.** NDCG-10 improvements reported over *old* baseline. $^{*}$ indicates a significant improvement over *old*, and $^{\dagger}$ over *recent* (p-value $< 0.05$, paired two-sided t-test). Values for *new* are averaged over all four days.

In the graphs in the lower row of Figure 3, we compare the relevance and freshness measured using several metrics such as precision (P), MAP, and NDCG using a cutoff of 5 and 10, respectively. These charts confirm that our hypothesis also holds for this analysis: Rankings produced from *new* data score higher than rankings produced from historic data.

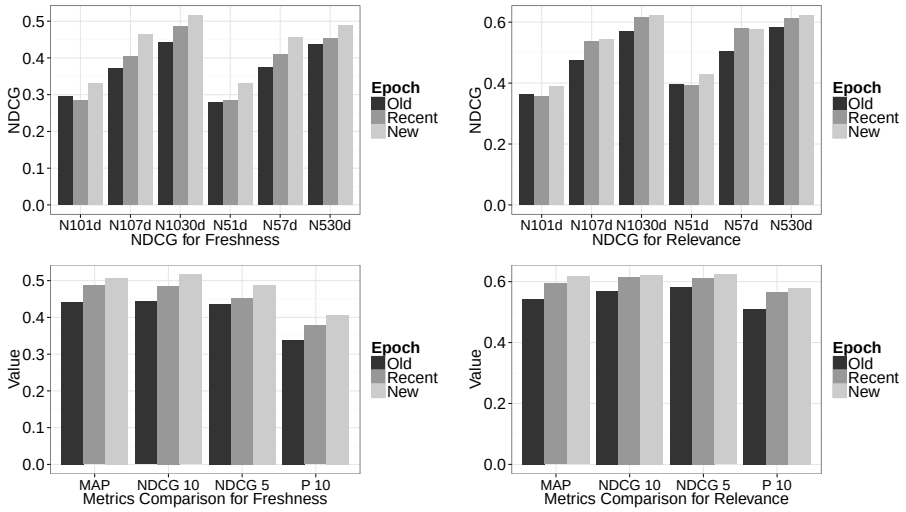



**Fig. 3.** Top: NDCG for the top 10/5 ranked entities for freshness and relevance
Bottom: Comparing several metrics for freshness and relevance for a 30 day window

| | $p(NotRel.\|\cdot)$ | $p(Rel.\|\cdot)$ |
|---|---|---|
| Super Recent | 0.15 | 0.85 |
| Very Recent | 0.21 | 0.79 |
| Recent | 0.50 | 0.50 |
| Reasonable | 0.31 | 0.69 |
| Outdated | 0.57 | 0.43 |
| NA or NJ | 0.45 | 0.55 |

(a) Relevance

| | $p(\cdot\|NotRel.)$ | $p(\cdot\|Rel.)$ |
|---|---|---|
| Super Recent | 0.02 | 0.11 |
| Very Recent | 0.05 | 0.14 |
| Recent | 0.23 | 0.17 |
| Reasonable | 0.14 | 0.25 |
| Outdated | 0.52 | 0.41 |
| NA or NJ | 0.05 | 0.03 |

(b) Freshness

**Table 6.** Distribution of recency across freshness and relevance values. "Super Related" and "Closely Related" collapsed into "Relevant" (Rel./Not Rel.).

**Fresh is Relevant** Table 6a shows the probabilities of different freshness labels conditioned to observing relevance and non-relevance, respectively. Relevance labels have been collapsed in the table, this is, we deemed the labels *Super Related* and *Closely Related* as relevant and all

other labels as not-relevant. The results suggest that freshness is a good indicator for relevance for the *Super Recent* and *Very Recent* categories. On the other hand, looking at Table 6b, we observe that relevance is not a good indicator for freshness. Intuitively this makes sense as it seems logical to assume that there are many more relationships between entities that may, although being relevant, not be of immediate importance in terms of recency. Overall, Pearson's correlation coefficient between labels is $0.28$, which indicates that there is only a slight correlation between them.

**More Data Is Better** In all but one case, having more data available to generate the rankings resulted in better performance in terms of relevance (top-right in Figure 3). Looking at the freshness evaluation, we observe a similar behavior (top-left in Figure 3) with the exception of the NDCG values computed using *new* data for the 7-day window, that for both, the top-10 and the top-5 ranks scored higher than the corresponding NDCG values for the 30-day window computed using "old" data. While this observation is consistent with the machine learning literature, it also shows that data that is more fresh can compensate in situations where only very little historic data can be collected.

**Performant Recent Models** In order to asses how well the GDBT models we employed generalize for unknown data, we used models of varying age to rank feature data generated from the most recent log data. The results of this comparison are shown in Figure 4: Using 20 day old data to train the models (Model Epoch = Old) yields worst performance for both freshness and relevance for all time windows, which suggests that the age of a trained model has an impact on performance. Comparing the performance achieved when using 10 day old (Model Epoch = recent) and current (Model Epoch = new) data, however, we can see that freshly trained models do not necessarily deliver better performance. This suggests, that while fresh data is important for ranking entities, the training of models is less time critical.

## 5 Conclusions & Future Work

We presented an evaluation of Sundog, a system for ranking relationships between entities on the web using a stream processing framework. Sundog is able to ingest large quantities of data at high rates (orders of

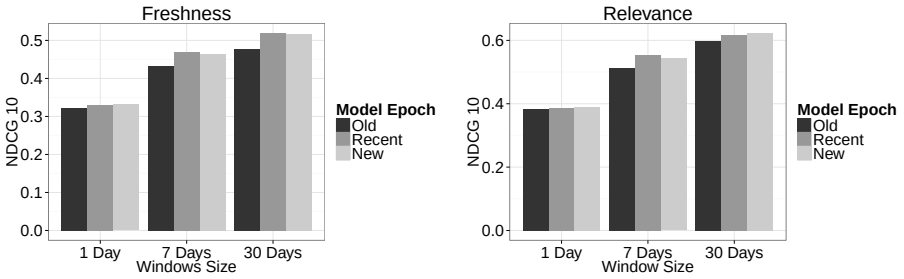



**Fig. 4.** Comparing the effect of the model age on freshness (left) and relevance (right). NDCG values obtained by applying new feature values on models of varying age.

magnitude more than a legacy batch-based system) and thus allows for adapting ranking into a live setting, where the ordering of elements (entities) displayed to the user might change with small time delays. This can be accomplished by inspecting relevance signals coming from query logs and updating feature values on the fly.

This architecture enabled us to investigate the tradeoff between data recency and relevance in a live setting, where rankings can change every day. We ran live experiments on four different days using real queries, generating rankings that were evaluated by professional human editors with regards to their *relevance* and *recency*. We trained different models, using old, recent and new data and reported their performance. It is apparent, that for sufficiently large time windows, *new* data always produces entity-rankings that are both, more fresh and also more relevant, with improvements reaching up to 24% in NDCG. We observed, that recency of input data can even compensate for reduced amounts of data, which is traditionally thought of as being a primary factor for the performance of machine learning models. Additionally, the ranking models we deployed are robust enough to be able to generalize well 10 days after they have been trained, even when feature values for query-entity pairs have changed over time. This suggests that while being able to process recent data is crucial, realtime re-learning does not impact performance as much (if at all).

While the source code of the system as well as the search log data used in the study are proprietary to Yahoo or cannot be released to the public for privacy reasons, we do provide a detailed description of the system and the data, which does allow for reproducibility of our results. For example, similar data sets that could be used are tweets from Twitter

or image tags from Flickr. In addition, Sundog is built using open source software, e.g. Apache Storm.

In future work, we will explore adaptions to the ranking model in more depth and also investigate, how freshness and relevance can be combined into one objective function. Currently, the models are learned by trying to maximize the relevance score. While recency and relevance are not necessarily two orthogonal performance characteristics, they can differ. The way in which one could combine these two aspects is not clear, yet. This, as well as an investigation of techniques with which recency and relevance can be independently measured without trained editors, remains future work. Additionally, we are interested in equipping the system with an online learner in order to make use of user feedback information (clicks) in real time. Finally, additional work on recency features is also necessary in order for the ranking models to be able to capture time dependent characteristics as for example concept shifts [25].

## Acknowledgments

We would like to thank B. Barla Cambazoglu, Alice Swanberg and her team, Derek Dagit, Dheeraj Kapur, Bobby Evans, Balaji Narayanan, and Karri Reddy for their invaluable support.

# References


[1] Krisztian Balog, Pavel Serdyukov, and Arjen P. de Vries. Overview of the trec 2011 entity track. In Ellen M. Voorhees and Lori P. Buckland, editors, *TREC*. National Institute of Standards and Technology (NIST), 2011.

[2] Roi Blanco, Berkant Barla Cambazoglu, Peter Mika, and Nicolas Torzec. Entity recommendations in web search. In *Proceedings of the 12th International Semantic Web Conference - ISWC '13*, pages 33–48. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-41338-4_3. URL http://dx.doi.org/10.1007/978-3-642-41338-4_3.

[3] Roi Blanco, Harry Halpin, Daniel M. Herzig, Peter Mika, Jeffrey Pound, Henry S. Thompson, and Thanh Tran. Repeatable and reliable semantic search evaluation. *Journal of Web Semantics*, 21:14–29, aug 2013. doi: 10.1016/j.websem.2013.05.005. URL http://dx.doi.org/10.1016/j.websem.2013.05.005.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, jul 1970. doi: 10.1145/362686.362692. URL http://dx.doi.org/10.1145/362686.362692.

[5] Wei Chen, Wynne Hsu, and Mong Li Lee. Modeling user's receptiveness over time for recommendation. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2484028.2484047. URL http://dx.doi.org/10.1145/2484028.2484047.

[6] Na Dai and Brian D. Davison. Freshness matters: In flowers, food, and web authority. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1835449.1835471. URL http://dx.doi.org/10.1145/1835449.1835471.

[7] Na Dai, Milad Shokouhi, and Brian D. Davison. Learning to rank for freshness and relevance. In *Proceedings of the 34th international ACM SIGIR conference on Research and development*

*in Information - SIGIR '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/2009916.2009933. URL http://dx.doi.org/10.1145/2009916.2009933.

[8] Anlei Dong, Yi Chang, Zhaohui Zheng, Gilad Mishne, Jing Bai, Ruiqiang Zhang, Karolina Buchner, Ciya Liao, and Fernando Diaz. Towards recency ranking in web search. In *Proceedings of the third ACM international conference on Web search and data mining - WSDM '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1718487.1718490. URL http://dx.doi.org/10.1145/1718487.1718490.

[9] Anlei Dong, Ruiqiang Zhang, Pranam Kolari, Jing Bai, Fernando Diaz, Yi Chang, Zhaohui Zheng, and Hongyuan Zha. Time is of the essence. In *Proceedings of the 19th International World Wide Web Conference - WWW '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1772690.1772725. URL http://dx.doi.org/10.1145/1772690.1772725.

[10] Jonathan L. Elsas and Susan T. Dumais. Leveraging temporal dynamics of document content in relevance ranking. In *Proceedings of the third ACM international conference on Web search and data mining - WSDM '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1718487.1718489. URL http://dx.doi.org/10.1145/1718487.1718489.

[11] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *2007 Conference on Analysis of Algorithms*, 2008. URL http://www.dmtcs.org/dmtcs-ojs/index.php/proceedings/article/view/dmAH0110.

[12] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, oct 2001. doi: 10.1214/aos/1013203451. URL http://dx.doi.org/10.1214/aos/1013203451.

[13] Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, feb 2002. doi: 10.1016/s0167-9473(01)00065-2. URL http://dx.doi.org/10.1016/S0167-9473(01)00065-2.

[14] Changsung Kang, Srinivas Vadrevu, Ruiqiang Zhang, Roelof van Zwol, Lluis Garcia Pueyo, Nicolas Torzec, Jianzhang He, and Yi Chang. Ranking related entities for web search queries. In

*Proceedings of the 20th international conference companion on World wide web - WWW '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/1963192.1963227. URL http://dx.doi.org/10.1145/1963192.1963227.

[15] Anagha Kulkarni, Jaime Teevan, Krysta M. Svore, and Susan T. Dumais. Understanding temporal query dynamics. In *Proceedings of the fourth ACM international conference on Web search and data mining - WSDM '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/1935826.1935862. URL http://dx.doi.org/10.1145/1935826.1935862.

[16] Thomas Lin, Patrick Pantel, Michael Gamon, Anitha Kannan, and Ariel Fuxman. Active objects: Actions for entity-centric search. In *Proceedings of the 21st International World Wide Web Conference - WWW '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2187836.2187916. URL http://dx.doi.org/10.1145/2187836.2187916.

[17] Donald Metzler, Rosie Jones, Fuchun Peng, and Ruiqiang Zhang. Improving search relevance for implicitly temporal queries. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '09*. Association for Computing Machinery (ACM), 2009. doi: 10.1145/1571941.1572085. URL http://dx.doi.org/10.1145/1571941.1572085.

[18] Peter Mika, Edgar Meij, and Hugo Zaragoza. Investigating the semantic gap through query log analysis. In *Proceedings of the 8th International Semantic Web Conference - ISWC '09*, pages 441–455. Springer Berlin Heidelberg, 2009. doi: 10.1007/978-3-642-04930-9_28. URL http://dx.doi.org/10.1007/978-3-642-04930-9_28.

[19] Jovan Pehcevski, James A. Thom, Anne-Marie Vercoustre, and Vladimir Naumovski. Entity ranking in wikipedia: utilising categories, links and topic difficulty prediction. *Journal of Information Retrieval*, 13(5):568–600, jan 2010. doi: 10.1007/s10791-009-9125-9. URL http://dx.doi.org/10.1007/s10791-009-9125-9.

[20] Jeffrey Pound, Peter Mika, and Hugo Zaragoza. Ad-hoc object retrieval in the web of data. In *Proceedings of the 19th International World Wide Web Conference - WWW '10*. Association

for Computing Machinery (ACM), 2010. doi: 10.1145/1772690.1772769. URL http://dx.doi.org/10.1145/1772690.1772769.

[21] Zhaochun Ren, Shangsong Liang, Edgar Meij, and Maarten de Rijke. Personalized time-aware tweets summarization. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2484028.2484052. URL http://dx.doi.org/10.1145/2484028.2484052.

[22] Milad Shokouhi and Kira Radinsky. Time-sensitive query autocompletion. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval - SIGIR '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2348283.2348364. URL http://dx.doi.org/10.1145/2348283.2348364.

[23] Roelof van Zwol, Lluís Garcia Pueyo, Mridul Muralidharan, and Börkur Sigurbjörnsson. Ranking entity facets based on user click feedback. In *2010 IEEE Fourth International Conference on Semantic Computing*. Institute of Electrical & Electronics Engineers (IEEE), sep 2010. doi: 10.1109/icsc.2010.33. URL http://dx.doi.org/10.1109/ICSC.2010.33.

[24] Roelof van Zwol, Lluís Garcia Pueyo, Mridul Muralidharan, and Börkur Sigurbjörnsson. Machine learned ranking of entity facets. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval - SIGIR '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1835449.1835662. URL http://dx.doi.org/10.1145/1835449.1835662.

[25] Peter Vorburger and Abraham Bernstein. Entropy-based concept shift detection. In *Sixth International Conference on Data Mining - ICDM '06*. Institute of Electrical & Electronics Engineers (IEEE), dec 2006. doi: 10.1109/icdm.2006.66. URL http://dx.doi.org/10.1109/ICDM.2006.66.

[26] Quan Yuan, Gao Cong, Zongyang Ma, Aixin Sun, and Nadia Magnenat-Thalmann. Time-aware point-of-interest recommendation. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval - SI-*

*GIR '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2484028.2484030. URL http://dx.doi.org/10.1145/2484028.2484030.

[27] Zhaohui Zheng, Hongyuan Zha, Tong Zhang, Olivier Chapelle, Keke Chen, and Gordon Sun. A general boosting method and its application to learning ranking functions for web search. In *Advances in Neural Information Processing Systems*, 2008. URL http://papers.nips.cc/paper/3305-a-general-boosting-method-and-its-application-to-learning-ranking-functions-for-web-search.pdf.